O'REILLY®

TURING 图灵程序设计丛书



高性能 iOS应用开发

High Performance iOS Apps

[美] Gaurav Vaish 著

梁士兴 郝田田 陈作君 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

译者介绍

梁士兴

毕业于北京航空航天大学，现任职美团－大众点评iOS高级技术专家。有多年的一线iOS开发经验，对于iOS应用性能有深入研究和独到见解。对iOS前沿技术、移动应用架构模式非常感兴趣。

郝田田

毕业于西安电子科技大学，现任职美团－大众点评iOS研发工程师，具有丰富的iOS软件开发经验，长期关注iOS领域前沿技术，曾参与数本iOS技术类书籍翻译工作。

陈作君

毕业于北京理工大学，现任职滴滴出行iOS研发工程师，多年大型移动端应用开发经验，热衷于开源项目建设，是一名技术书籍翻译的爱好者。

图灵程序设计丛书

高性能iOS应用开发

High Performance iOS Apps

[美] Gaurav Vaish 著
梁士兴　郝田田　陈作君　译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（C I P）数据

高性能iOS应用开发 / (美) 高拉夫·瓦依希
(Gaurav Vaish) 著 ; 梁士兴, 郝田田, 陈作君译. --
北京 : 人民邮电出版社, 2017.4
（图灵程序设计丛书）
ISBN 978-7-115-45120-0

Ⅰ. ①高… Ⅱ. ①高… ②梁… ③郝… ④陈… Ⅲ.
①移动终端－应用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字(2017)第051216号

内 容 提 要

性能对用户体验有着至关重要的影响。本书将介绍对用户体验产生负面影响的各个方面，并概述如何优化 iOS 应用的性能。全书共 5 个部分，主要从性能的衡量标准、对应用至关重要的核心优化点、iOS 应用开发特有的性能优化技术以及性能的非代码方面，讲解了应用性能的优化问题。本书的主要目的是展示如何从工程学的角度编写最优代码。

本书适合已经具有 Objective-C 和 iOS 实践经验的开发人员阅读。

◆ 著　　　　[美] Gaurav Vaish
译　　　　梁士兴　郝田田　陈作君
责任编辑　朱　巍
执行编辑　杨　婷
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：23.75
字数：567千字　　　　2017年 4 月第 1 版
印数：1－3 500册　　　2017年 4 月北京第 1 次印刷
著作权合同登记号　图字：01-2017-0540号

定价：89.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第 8052 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第二部分 核心优化

第三部分 iOS 性能

第四部分　代码之外

第五部分 iOS 9

译者序

移动互联网经历了近五年的高速发展后，增长速度逐步趋于平缓。疯狂过后，逐渐回归理性。尽管红利不再，但其整体规模已经得到了极大的发展，手机和 App 已经成为了人们日常生活中很重要的一部分。在这个阶段，移动互联网的战争会变得更加惨烈。冷静下来的人们不禁会思考一个问题：如何让我们的 App 更有竞争力？

排除提供服务的能力差异（这通常与 App 本身无关），体验做到极致才更有机会捕获用户的芳心。如何把体验做得更好呢？一方面，“颜值”很重要，在这个“看脸”的年代，App 的颜值太差一定会被用户嫌弃。另一方面，性能更重要，更高的性能意味着更短的等待时间、更平滑流畅的体验、更低的内存使用和更少的电量消耗。对于每个程序员来说，“我的 App 性能最好”，绝对是一件值得炫耀的事情。但要想真正做到这一点，却十分困难。

本人就职于美团大众点评的酒店旅游事业群，常年奋战在一线，专注于 iOS App 的开发和优化，在性能优化方面也积累了大量的实战经验。工作之余，一直有这样一种想法：如果有一本书能够系统地阐述 iOS App 性能优化的方方面面，一定会对我和我们的团队有巨大的帮助！第一次见到这本书，我就立即被它的内容吸引了。书中涵盖了 iOS App 性能优化的方方面面，既有广度，又有深度。书中介绍的知识点，可以非常容易地应用到实际的项目中；很多的技术点，和我们之前所做的优化简直是不谋而合，大有相见恨晚之感。本书凝聚了作者在性能优化方面付出的大量心血，值得每一位期望进阶的工程师深入地阅读和学习。能够接手本书的翻译任务，对我来说既兴奋又充满压力。尽可能快地把这样一部优秀作品的中文译本高质量地交付给读者，对我来说是件充满激情和挑战的事情。

为了做好本书的翻译工作，我们克服了许多困难。首先是时间，我们需要利用业余时间和尽可能多的碎片时间进行本书的翻译工作，经常深夜还看到小伙伴们仍然在奋笔疾书。不仅如此，我还有些许忧虑，担心自己把握不好原著恰到好处的笔锋，不能有效地将这样一部优秀的作品呈现在读者面前。因此，我们对这次翻译格外用心，与几位合作者一起查阅了大量相关资料，力求做到专业词汇准确权威，将原书的精华呈现给每一位读者。

现在，我怀着忐忑的心情，将此译著呈现给读者，渴望得到读者认可，更渴望与读者成为朋友。如果有任何问题和建议，请与我联系（liangshixing@gmail.com），让我们一起探讨，共同进步。

感谢北京图灵文化发展有限公司的编辑部主任朱巍女士，感谢你对我们的信任和支持。感谢编辑杨婷和谢婷婷，感谢你们提出的大量宝贵意见。感谢（美团大众点评）酒店旅游事业群负责人陈亮，感谢你建立了这支充满技术热情的移动团队。感谢另外两位译者，团队的同事郝田田和多年的好友陈作君，感谢大家几个月以来的共同努力。最后，还要感谢我的家人，他们给予我理解与包容。

梁士兴

2016 年 11 月于北京

谨以此书献给我的儿子 Agryav Vaish

前言

你可能已经开发过一个令人赞叹的 iOS 应用，或者正在开发。如果应用整体运行良好但存在一些缺陷，那么用户可能不会给予五星好评，这些缺陷甚至会影响发布。

一些问题是用户可以直接发现的，例如，用户点击表格视图的某一项时出现抖动现象，应用的流量消耗过多或耗电量巨大。但是这些问题可能发生在更深的层面。

优化应用的性能是一项永无止境的工作，尤其在应用的新特性、操作系统版本、第三方库和设备配置层出不穷的情况下。而这些只是让开发者关注应用性能的一小部分内容。

一项研究表明，如果应用无法在三秒内加载启动，那么约四分之一的用户将弃用此应用，约三分之一的用户会将这段令人不快的经历转告他人。

用户希望应用运行快速、响应迅速且不占用过量的资源。本书将介绍对用户体验产生负面影响的各个方面，并概述如何优化应用的性能。

本书读者

如果你写过 iOS 应用并发布到了 App Store，那么你的隐含目标是让应用更好、更快、更流畅，毫无疑问，你的最终目标是让应用为用户所喜爱。如果你正在寻找实现这个目标的方法，那么本书正是为你而准备的。

你应该已经具有 Objective-C 和 iOS 的实践经验。虽然在必要时为内容完整起见会介绍一些基本原理，但本书不会讨论如何使用 Objective-C 或如何进行 iOS 的入门开发。

本书撰写初衷

第一代 iOS 和 iPhone 1 于 2007 年 6 月推出。在早期版本中，开发人员忙于清理代码，为更多用户发布应用。随着硬件、操作系统、网络以及整体生态系统的不断提升，新的 UI 和工程设计模式不断涌现，应用在功能、稳定性和性能方面逐渐成熟。

通常情况下，性能是后来才会考虑的事情。从某种程度上来说，我认同这种观点，毕竟最

重要的是先完成功能，而不是担心性能。在软件开发周期的早期考虑性能通常被称为过早优化，但是，当糟糕的性能表现暴露时，问题就太严重了。

本书的主要目的是向读者展示如何从工程学的角度编写最优代码。

本书并非通过计算机理论科学、数据结构和算法来更快地执行程序。你可以找到很多关于这些主题的图书。本书涵盖了实现应用的最佳实践，即使在非理想条件（低存储空间、不良网络、低电量等）下，让用户仍然可以有效地使用应用，并乐于使用应用。通常而言，你不可能优化所有的参数，但考虑有效因素可以实现最佳平衡。

本书预览

本书共由五个部分组成，每一个部分由一章或多章根据特定的主题组成。每章开头会有简短的摘要说明。

第一部分概述如何衡量性能。第 1 章讨论可优化的方面，并概述跟踪应用性能时需要衡量的参数。

第二部分回顾对应用至关重要的核心优化点。第 2 章讨论内存管理问题，其中描述了内存管理模型和对象引用类型，还讨论了影响内存消耗的设计模式的最佳实践，即单例和依赖注入。第 3 章讨论电量及可以最大限度减少其消耗的技术。第 4 章为并发编程概述，其中描述了各种有效方法，并提供了对比分析。

第三部分涉及 iOS 应用开发特有的性能优化技术。第 5 章深入探讨应用的生命周期，详细介绍了如何利用生命周期事件来确保资源的高效使用。第 6 章专门阐述针对 UI 的优化技术。第 7 章和第 8 章分别讨论网络和数据共享。第 9 章深入探讨了应用的安全问题，了解增强安全性会对应用的运行效率产生哪些负面影响，以及如何在两者间实现有效的平衡。

第四部分讨论性能的非代码方面。第 10 章的内容涉及测试，特别是性能测试，此外还讨论了持续集成和测试自动化。第 11 章概述开发过程中用于衡量性能的工具。第 12 章讨论埋点和分析，以及如何从生产环境的应用中收集与性能相关的数据。

第五部分重点介绍 iOS 9 及 iOS 10。第 13 章概述 iOS 9 的变化，并从性能角度分析它们是如何影响你编写的代码的。第 14 章概述了 iOS 10 的变化。

本书提供了可运行的代码段。其中部分代码段可以原样使用，或只需要在应用中进行小修改。其他代码段可能需要进一步调整，以适应你自己的应用。

每章还提供了与该主题相关的一组最佳实践。在单一应用中可能无法始终遵循所有的最佳方案，可根据应用的具体要求对优化点进行取舍。

在线资源

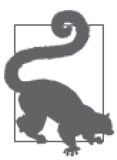
本书涉及许多在线博客、文章、教程和其他参考资料，并在合适的地方提供了这些参考文献的链接。如果你发现有遗漏之处，请随时与出版社或作者联系。

本书还引用了几个应用的截图。应用的版权归其拥有者所有，本书使用屏幕截图仅用于教育和说明。

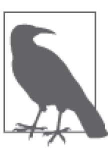
排版约定

本书使用了下列排版约定。

- 楷体
  表示新术语或重点强调内容。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 等宽斜体（*`Constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/gvaish/high-performance-ios-apps 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。例如："*High Performance iOS Apps* by Gaurav Vaish (O'Reilly). Copyright 2016 Gaurav Vaish, 978-1-491-91100-6."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920034506.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

独立写一本书可能并不多见。

非常感谢雅虎的杰出架构师 Daryl Low 与我密切合作，我们一起从头开始开发了货币化 SDK。他为书中的部分章节提供了指导意见。与他一起通过应用原型测试性能极限并发现错误的根本原因一直是有趣和令人兴奋的工作。

感谢雅虎移动 DevOps 工程负责人 Walter Pezzini。他在我了解持续集成和交付流程时提供了关键见解，并帮助我了解构建高质量系统所需要的内容。

作者很容易假设读者已事先了解一些知识，但有时事实并非如此。非常感谢 Chris Devers、Laura Savino 和 Niklas Saers 的意见，他们让我明确需要澄清的领域。同样感谢他们提供反馈以帮助提高整体内容质量。

感谢我的妻子 Renu Chandel，她不断鼓励我完成本书。如果不是因为她，本书也不会完成。还要感谢所有的咖啡！

最后，感谢 O'Reilly Media 出版了本书。

本书的出版是大家共同努力的成果。谢谢大家。

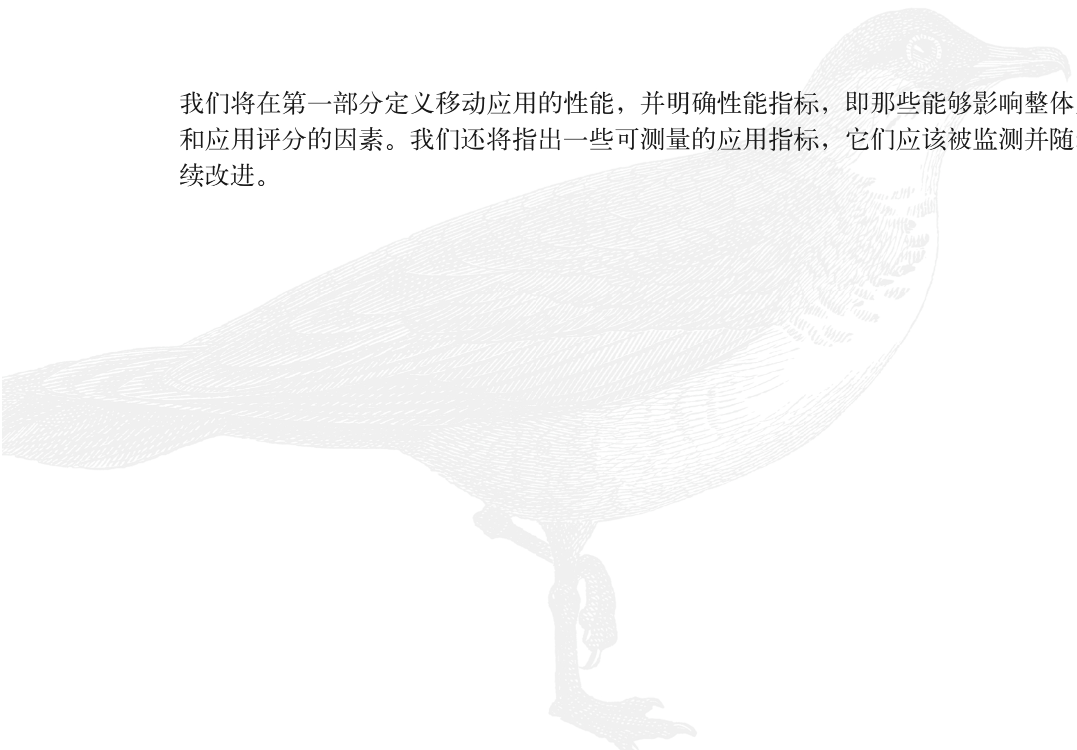
电子书

扫描如下二维码，即可购买本书电子版。

第一部分

开始

我们将在第一部分定义移动应用的性能，并明确性能指标，即那些能够影响整体用户体验和应用评分的因素。我们还将指出一些可测量的应用指标，它们应该被监测并随着时间持续改进。



第 1 章

移动应用的性能

本书假设你是 iOS 开发人员，有长期开发原生 iOS 应用的经验，并且希望能够从众人中脱颖而出，跻身于顶尖开发人员之列。

参考以下统计数据。[1]

- 应用首次工作出错以后，79% 的用户只会再重试一两次。
- 当应用载入时间超过 3 秒时，25% 的用户会放弃使用该应用。
- 31% 的用户会将糟糕的体验转告他人。

这些数据强调了性能对应用的重要性。应用能否被用户所认可不仅仅取决于其功能，还取决于当与用户交互时，应用能否提供流畅的体验。

几乎完成任意特定任务的应用都能在 App Store 中找到大量的替代品。但用户只会坚持使用其中的某一款。被选中的这一款要么无可取代，要么极少出现故障且性能格外出众。

性能会受许多重要因素所影响，这些因素包括内存消耗、网络带宽效率以及用户界面的响应速度。我们先概述不同类型的性能特征，然后再对它们进行测量。

1.1 定义性能

从技术视角严格来说，性能是非常模糊的术语。当一个人说“这是个高性能的应用”时，其实我们无从判断他说的是什么。他是说应用消耗的内存少？应用节约了网络流量？还是说应用使用起来非常流畅？总而言之，高性能有着多重的含义和丰富的解释方式。

注 1：Hewlett Packard Enterprise Software Solutions,“3 keys to a 5-star mobile experience”.(http://www.slideshare.net/HPESoftwareSolutions/3-keystoa5starmobileexperience)

性能可以和我们后续讨论的多个要点关联起来。（需要测量和监控的）*性能指标*是其中的一个关注点，（实际上收集数据的）*测量*是另一个关注点。

我们将在第 11 章深入探索测量的过程。提高工程*参数*的使用率是本书第二部分和第三部分的重点难题。

1.2 性能指标

性能指标是面向用户的各种属性。每个属性可能是一个或多个可测量工程*参数*的一个要素。

1.2.1 内存

内存涉及运行应用所需的 RAM 最小值，以及应用消耗的内存平均值和峰值。最小内存值会严重限制硬件，而更高的内存平均值和峰值意味着更多的后台应用会被强制关闭。

同时还要确保没有泄漏内存。随时间流逝而持续增长的内存消耗意味着，应用很可能会因为内存不足的异常而崩溃。

我们会在第 2 章中对内存进行深入讨论。

1.2.2 电量消耗

在编写高性能代码时，电量消耗是一个需要重点处理的重要因素。就执行时间和 CPU 资源的利用而言，我们不仅要实现高效的数据结构和算法，还需要考虑其他的因素。如果某个应用是个电池黑洞，那么一定不会有人喜欢它。

电量消耗不仅仅与计算 CPU 周期有关，还包括高效地使用硬件。除了要实现电量消耗最小化，还要确保不会影响用户体验。

我们将在第 3 章讨论这个问题。

1.2.3 初始化时间

应用在启动时应执行*刚好够用*的任务以完成初始化，从而满足用户的使用需求。执行这些任务消耗的时间就是应用的初始化时间。*刚好够用*是一个开放式用语——正确的平衡点取决于应用的需要。

在首次使用应用时创建对象并进行初始化是一个合理的选择，例如，直到需要使用对象时才创建对象。这种方式被称为*惰性初始化*。这是一种很好的策略，但也要考虑不能让用户总是在执行后续任务时等待。

下面列举了你可能想在应用初始化阶段执行的一些动作，排名不分先后。

- 检查应用是否为首次启动。
- 检查用户是否已经登录。
- 如果用户已经登录，尽可能地载入之前的状态。
- 连接服务器以拉取最新的变更。

- 检查应用是否由某个深层链接唤起。如果是，还需要载入深层链接相应的 UI 和状态。
- 检查是否存在应用上次启动时挂起的任务，需要时恢复它们。
- 初始化后续需要使用的对象和线程池。
- 初始化依赖项（如对象关系映射、崩溃报告系统和缓存）。

这个列表可能会迅速变长，并且很难决定哪些条目一定要在启动时执行，哪些可以延后几毫秒再执行。

我们将在第 5 章探讨这个问题。

1.2.4 执行速度

一旦启动应用，用户总是希望它可以尽可能快地工作。一切必要的处理都应该在尽可能短的时间内完成。

例如，在照片应用中，用户通常希望看到调整亮度或对比度等简单效果的实时预览效果。因此，相应的处理需要在几毫秒内完成。

这可能需要本地计算的并行处理技术或能够将复杂任务分发到服务器。我们将在第 4 章和第 6 章介绍这些主题，并在第 7 章和第 11 章介绍相关工具。

1.2.5 响应速度

每个应用都应该快速地响应用户交互。在应用中所做的一切优化和权衡最终都应该体现在响应速度上。

App Store 中有许多应用可以完成相似或相关的任务。这为用户提供了很大的选择空间，而用户基本都会选择响应最快的应用。

第 4 章介绍了用并行处理技术优化本地执行。第 5 章和第 6 章介绍了实现流畅交互的最佳实践。第 10 章将探索如何对应用进行测试。

1.2.6 本地存储

针对任何在服务器上存储数据或通过外部来源刷新数据的应用，开发人员应该对本地存储的使用有所规划，以便应用具备离线浏览的能力。

例如，用户都希望邮件应用能够在无网络或设备离线的情况下浏览历史邮件。

同样，新闻应用也应该可以在离线模式下显示最近更新的新闻，并标记出每条新闻是否已读。

然而，从本地存储中载入和同步数据应该迅速、便捷。这不仅需要选择要在本地缓存的数据和要优化的数据结构，还需要提供一系列的配置选项并确定数据同步的频率。

如果你的应用使用了本地存储，那么请提供一个清除数据的选项。遗憾的是，市场上的大部分应用都没有提供此选项。更让人烦恼的是，一些应用竟然会消耗数百兆的存储空间。用户会频繁地卸载这些应用来回收本地存储。这会导致糟糕的用户体验，从而威胁应用的成功。

如图 1-1 所示，超过 12GB 的空间已经被使用了，留给用户的内存还有 950MB。其实，大部分的数据可以安全地从本地删除。这些应用应该提供清理缓存的选项。

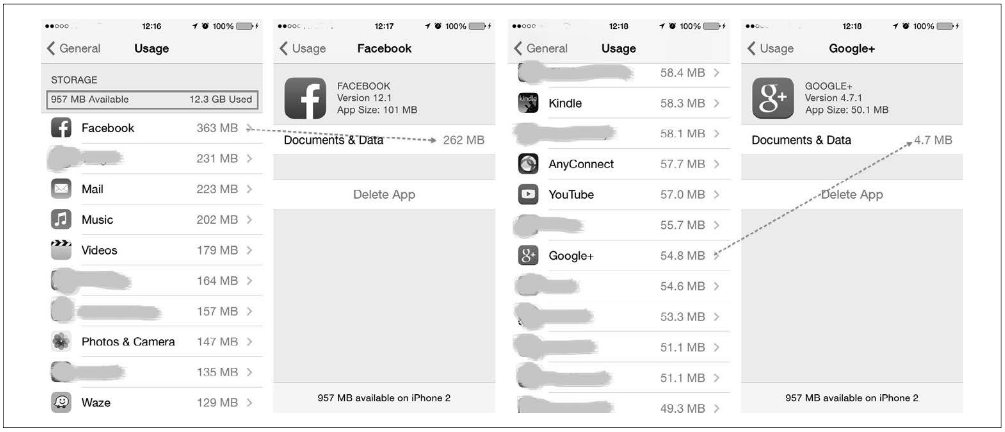


图 1-1：磁盘使用状况

> 一定要向终端用户提供清空本地缓存的选项。
> 如果用户开启了 iCloud 的备份功能，那么应用的数据将会消耗用户的存储限额，请谨慎使用。

第 7 章、第 8 章和第 9 章会介绍本地存储相关的话题。

1.2.7 互操作性

用户可能会使用多个应用来完成某个任务，这就需要这些应用直接提供互操作的能力。例如，一个相册可能需要一个幻灯片应用来实现最佳的浏览体验，但需要另一个应用来编辑照片。其中浏览照片的应用要能够将照片发送到编辑器，并接收编辑后的图片。

iOS 为实现应用间的互操作和数据共享提供了多种机制，其中包括 `UIActivityViewController`、深层链接、`MultipeerConnectivity` 框架，等等。

为深层链接定义良好的 URL 结构与编写优异的代码来解析 URL 同样重要。类似地，使用共享对话框共享数据时，精确识别用于分享的数据非常重要，同时，在处理不同数据源传入的数据时还要注意安全隐患。

如果某个应用向附近设备共享数据时需要花费很长时间准备数据，那么用户体验就会非常糟糕。

我们会在第 8 章中讨论这些内容。

1.2.8　网络环境

移动设备会在不同网络环境下使用。为了确保能够提供最好的用户体验，你的应用应当适应各种网络条件：

- 高带宽稳定网络
- 低带宽稳定网络
- 高带宽不稳定网络
- 低带宽不稳定网络
- 无网络

为用户提供进度指示或错误信息是相对合理的方式，无尽的等待或崩溃则让人无法接受。

图 1-2 的屏幕截图展示了向终端用户传递信息的不同方式。TuneIn 应用显示了已经缓冲的信息流大小，以此告诉终端用户还需要等待多久才可以播放音乐。MoneyControl 和 Bank of America 等其他应用仅提供了不明确的进度条，这在非流式应用中是更为常见的样式。

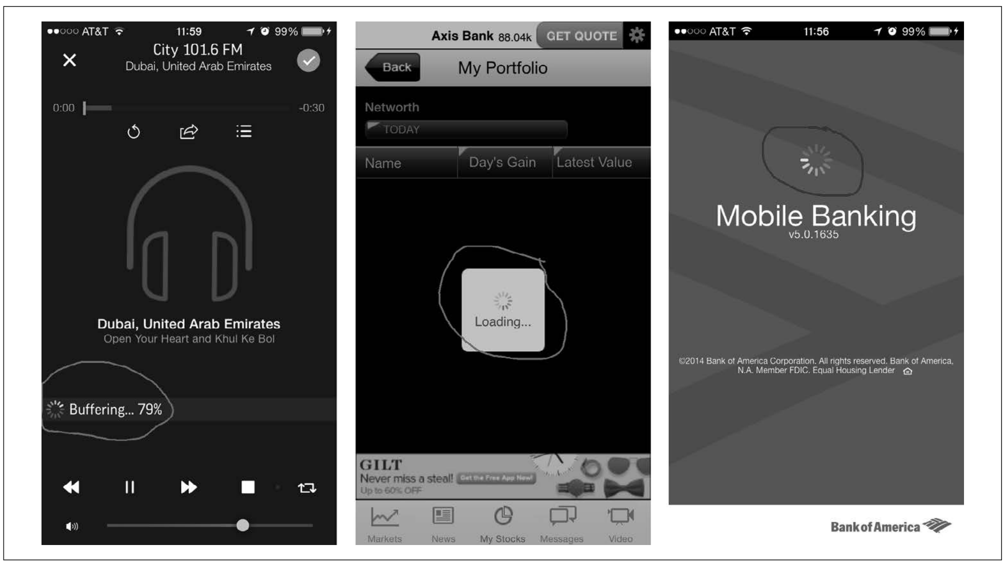


图 1-2：因为网络环境差或数据量大而显示的不同提示信息

我们将在第 7 章中深入探讨此话题。

1.2.9　带宽

人们会在不同的网络条件下使用自己的移动设备，网速从每秒数千字节到每秒数十兆字节。

因此，带宽的优化使用是定义应用质量的另一个关键参数。此外，在高带宽网络下运行一个基于低带宽网络开发的应用可能会产生完全不同的结果。

2010 年左右，我和我的团队正在印度开发一款应用。由于处于低带宽网络，应用的本地初始化速度要比从服务器端载入资源快得多，于是我们针对这种情况进行了优化。

然而，当这款应用投入韩国市场时，我们对它进行了测试，结果却让人大跌眼镜。之前所进行的所有优化几乎毫无意义，我们不得不重写了大部分可能导致资源和数据冲突的相关代码。

为提高性能所做的设计并非每次都能如愿，也可能会导致相反的效果。

第 7 章包含了优化使用带宽的最佳实践。

1.2.10　数据刷新

即使没有提供离线浏览能力，你仍然可以从服务器端周期性地刷新数据。刷新的频率和每次传输的数据量将决定数据传输的总量。如果传输的字节数过大，那用户必然会快速耗尽自己的流量计划。当流量消耗大到一定程度时，你的应用很可能会流失用户。

在 iOS 6.x 或更低版本中，在后台运行的应用不能刷新数据。从 iOS 7 开始，应用可以在后台周期性地刷新数据。对于在线聊天类应用，持久的 HTTP 连接或原生 TCP 连接可能会非常有用。

第 5 章和第 7 章会介绍这部分内容。

1.2.11　多用户支持

家庭成员间可能会共享移动设备，或者一个用户可能会拥有同一应用的多个账号。例如，兄弟姐妹间可能会共享一个 iPad 来玩游戏。再比如，家庭成员可能会在旅游时配置一个设备来查收全家人的电子邮件，以减少漫游费用，尤其是在境外旅游时。类似地，一个人也可能会配置多个电子邮件账号。

是否支持多个并发用户取决于产品的需要。一旦决定提供此类功能，请参考以下准则。

- 添加新用户应尽可能高效。
- 在不同用户之间更新应尽可能高效。
- 在不同用户之间切换应尽可能高效。
- 用户数据的界限应该简洁且没有 bug。

图 1-3 展示了两个提供了多用户支持的应用。左边和右边分别展示了 Google 和 Yahoo 应用的账号选择功能。

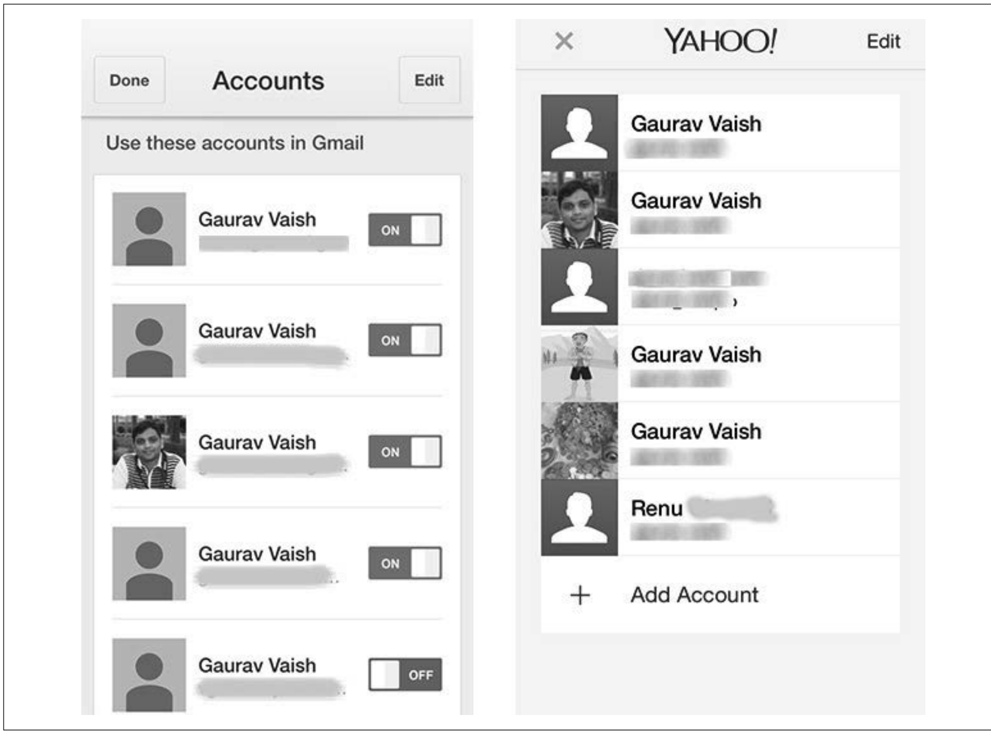


图 1-3：Google 和 Yahoo 应用都提供了多用户支持

第 9 章将介绍如何在应用支持多用户的同时保障安全以及其他内容。

1.2.12　单点登录

如果你已经创建了多个允许或需要登录的应用，那么支持单点登录（single sign-on，SSO）是非常棒的选择。如果用户登录了一个应用，只需要点击一次，就可以登录到其他的应用中。

这个过程不仅需要支持跨应用的数据共享，还需要分享状态、跨应用同步等。例如，如果用户注销了其中某个应用，则通过 SSO 登录的所有其他应用也应能注销掉。

此外，应用之间的同步应该是安全的。

第 9 章将会介绍这部分内容。

1.2.13　安全

安全对移动应用来说是最重要的，因为敏感信息可能会在应用间共享。因此，对所有通信以及本地数据和共享数据进行加密就显得尤为重要了。

实现安全需要更多的计算、内存和存储，但这与最大化运行速度、最小化内存和存储使用的目标相冲突。

因此，你需要在安全和其他因素之间进行权衡。

引入多个安全层会影响性能，并对用户体验造成可感知的负面影响。如何设定安全的基线需要参考对用户群体的统计分析。此外，硬件在其中扮演了重要的角色：选择会因为不同设备的计算能力而有所不同。

第 9 章将会深入介绍安全。

1.2.14 崩溃

应用可能会而且确实会崩溃。过度优化会导致崩溃。同样，使用原始 C 代码也可能会导致崩溃。

高性能的应用不仅应尽可能地避免崩溃，还应该在崩溃发生时优雅地恢复，尤其是在进行某个操作的过程中发生崩溃时。

第 12 章会深入讨论崩溃报告、检测和分析。

1.3 应用性能分析

我们在前面讨论过一些参数，通过测量它们来分析应用的方式有两种：采样和埋点。接下来我们将逐一介绍。

1.3.1 采样

顾名思义，采样（或基于探测点的性能分析）是指以一定的周期间隔采集状态，这通常需要借助工具。我们将在 11.2 节中介绍这些工具。由于不会干扰应用的执行，因此采样可以很好地提供应用的全景图。采样的不足之处在于它不能返回 100% 精确的细节。如果采样的频率是 10 毫秒，那么你就无法得知在探测点之间的 9.999 毫秒内发生了什么。

> 采样可以作为初始的性能调研手段，并可用于跟踪 CPU 和内存的使用情况。

1.3.2 埋点

通过修改代码，记录细节信息的埋点能够提供比采样更加精确的结果。你既可以在关键部分主动埋点，也可以在性能分析或处理用户反馈时有针对性地埋点，以便解决问题。1.4.3 节将深入讨论这一过程。

> 因为埋点需要注入额外代码，所以它一定会影响应用的性能，对内存或速度（或同时对二者）造成损害。

1.4 测量

现在，我们已经确定了需要测量的参数，并且研究了测量所需要的不同类型的分析。我们先简单了解一下如何实现测量。

通过测量性能并找出真正存在问题的地方，你可以避免掉入过早优化的陷阱。高德纳曾经这样描述过早优化：

> 真正的问题在于，程序开发人员为提升程序效率在错误的方向和时间点浪费了太多时间；过早优化是编程领域的万恶（至少是绝大多数的恶）之源。[2]

1.4.1 设置工程与代码

接下来，我们将建立一个工程，以便在开发和生产阶段测量已经定义好的参数。针对工程配置、安装和代码实现共有三类任务。

- 构建与发布
  确保能够轻松地构建和发布应用。
- 可测试性
  确保你的代码能够同时在模拟数据和真实数据之上工作，其中包括能够模拟真实场景的隔离环境。
- 可跟踪性
  确保你能够通过明确问题发生的位置和代码行为来处理错误。

接下来将逐一讨论这些设置项。

1. 构建与发布

构建和发布是直到最近才出现的话题。好在由于对灵活和敏捷的强烈需求，系统和工具得到了改进。改进后的系统和工具现在可以加速拉取依赖信息，加速构建和发布用于测试或企业分发的产品，也可以为公众发布而提高提交文件到 iTunes Connect 的速度。

2000 年，Joel Spolsky 在其发表的一篇博文（http://www.joelonsoftware.com/articles/fog0000000043.html）中提出了一个问题："你能（从源码）一键构建自己的应用吗？"这个问题现在依然成立，且问题的答案很可能会决定你在发现缺陷或瓶颈后改进质量和解决性能问题的速度。

基于 Ruby 语言实现的 CocoaPods（https://cocoapods.org）实际上是 Objective-C 和 Swift 工程的依赖管理器。[3]CocoaPods 与 Xcode 命令行工具相集成，可用于构建与发布。

2. 可测试性

每个应用都包含多个协同工作的组件。一个设计良好的系统应该遵循低耦合和高内聚，并

注 2：Donald Knuth, "Computer Programming as an Art" (https://en.wikiquote.org/wiki/Donald_Knuth#Computer_Programming_as_an_Art_.281974.29)

注 3：在撰写本书时，以 CocoaPods 发布的绝大多数对象都是用 Objective-C 编写的，毕竟 Swift 比 Objective-C 要新得多。

允许替换任意或全部组件的依赖。

可以通过模拟依赖项目对每个组件进行隔离测试。一般来说，测试有两种类型。

- 单元测试
  验证每个代码单元在隔离环境下的操作。常见的做法是，在特定的环境中用不同的输入数据反复地调用一些方法，以评估代码的表现。

- 功能测试
  验证组件在最终集成的安装包中的操作。可以在软件的最终发布版本中验证，也可以在某个为测试而构建的参考应用中验证。

我们将在第 10 章中深入讨论测试。

3. 可跟踪性

在开发阶段，埋点可以帮助我们确定性能优化的优先级、提高对问题现场的还原能力，并提供更多的调试信息。崩溃报告专注于从软件的产品版本中收集调试信息。

1.4.2 设置崩溃报告

崩溃报告系统收集用于分析应用的调试日志。市面上有数十种崩溃报告系统。本书选用了 Flurry（http://www.furry.com），但这并不代表我对其他系统有任何偏见。选用 Flurry 的主要原因是，只用一个 SDK 就可以同时实现崩溃报告和埋点。我们将在第 12 章深入介绍埋点。

要想使用 Flurry，需要在 www.furry.com 中建立一个账户，得到一个 API 密钥，然后下载并设置 Flurry SDK。例 1-1 展示了初始化 Flurry 的代码。

例 1-1 在应用委托中配置崩溃报告

```
#import "Flurry.h"

- (BOOL)application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

  [Flurry setCrashReportingEnabled:YES];

  [Flurry startSession:@"API_KEY"]; ❶
}
```

❶ 用账户关联的 API 密钥（可以在 Flurry 的 Dashboard 中找到）替换 `API_KEY`。

> 崩溃报告系统会用 `NSSetUncaughtExceptionHandler` 方法设置全局的异常处理器。使用自定义的处理器则会失效。
> 如果希望继续使用自己的处理器，那么你需要在崩溃报告系统初始化之后再进行设置。此外，还可以通过 `NSGetUncaughtExceptionHandler` 方法得到崩溃报告系统所设置的处理器。

1.4.3 对应用埋点

对应用进行埋点是了解用户行为的一个重要步骤，但更重要的目的是识别应用的关键路径。注入特定的代码以记录关键指标是提升应用性能的重要步骤。

> 对依赖进行抽象化和封装是个好主意。这样就可以在最后再进行切换，甚至可以在作出最终决定之前同时使用多个系统。这在项目处于评估阶段且存在多个备选方案时尤为有用。

如例 1-2 所示，我们将增加一个名为 `HPInstrumentation` 的类来封装埋点。现在，我们用 `NSLog` 登录控制台，并向服务器发送细节。

例 1-2　`HPInstrumentation` 类封装了埋点 SDK

```
//HPInstrumentation.h
@interface HPInstrumentation : NSObject

+(void)logEvent:(NSString *)name;
+(void)logEvent:(NSString *)name withParameters:(NSDictionary *)parameters;

@end

//HPInstrumentation.m
@implementation HPInstrumentation

+(void)logEvent:(NSString *)name
{
    NSLog(@"%@", name);
    [Flurry logEvent:name];
}

+(void)logEvent:(NSString *)name withParameters:(NSDictionary *)parameters
{
    NSLog(@"%@ -> %@", name, params);
        [Flurry logEvent:name withParameters:parameters];
}

@end
```

先在应用生命周期的三个关键阶段进行埋点（见例 1-3）：

- 每当应用进入前台，`applicationDidBecomeActive:` 方法会被调用
- 每当应用进入后台，`applicationDidEnterBackground:` 方法会被调用
- 如果应用收到低内存警告，`applicationDidReceiveMemoryWarning:` 方法会被调用

出于娱乐的目的，我们在 `HPFirstViewController` 中添加一个按钮，点击该按钮将导致应用崩溃。

例 1-3　在应用委托中的基础埋点

```
- (void)applicationDidBecomeActive:(UIApplication *)application
{
```

```
    [HPInstrumentation logEvent:@"App Activated"];
}

- (void)applicationDidEnterBackground:(UIApplication *)application
{
    [HPInstrumentation logEvent:@"App Backgrounded"];
}

- (void)applicationDidReceiveMemoryWarning:(UIApplication *)application
{
    [HPInstrumentation logEvent:@"App Memory Warning"];
}
```

我们将这些事件分别命名为 `App Activated`、`App Backgrounded` 和 `App Memory Warning`。你可以选择自己喜欢的任意名称，也可以使用数字。

> 埋点不应该取代日志。日志可以非常详细。但因为向服务器报告时会消耗网络资源，所以你应该尽可能少地埋点。
>
> 因此，只对你和其他工程或产品团队的成员感兴趣的事件进行埋点是非常重要的。（这些事件要包含足够多的数据以满足重要报告的需要。）
>
> 埋点和过度埋点之间并没有清晰的分界线。一开始应仅对少量报告进行埋点，然后随着时间的推进逐步增加埋点的覆盖率。

接下来添加一个 UI 控件，让它能够触发一个崩溃，这样我们就可以看到崩溃报告。

图 1-4 展示了崩溃按钮的 UI。例 1-4 展示了连接 `Touch Up Inside` 事件和 `crashButtonWasClicked:` 方法的代码。

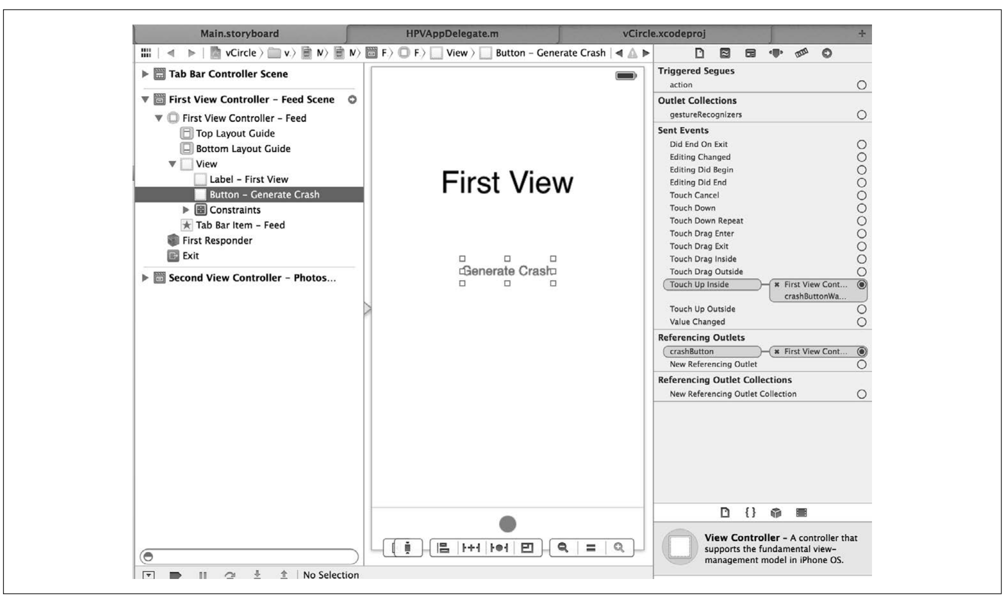


图 1-4：在故事板中添加 Generate Crash 按钮

例 1-4　抛出异常使得应用崩溃

```
- (IBAction)crashButtonWasClicked:(id)sender
{
    [NSException raise:@"Crash Button Was Clicked" format:@""];
}
```

让我们与应用进行交互，从而产生一些事件。

(1) 安装并运行应用。
(2) 将应用切换到后台。
(3) 将应用切换到前台。
(4) 多次重复第 2 步和第 3 步。
(5) 点击 Generate Crash 按钮，导致应用崩溃。
(6) 再次运行应用。直到此时崩溃报告才会实际发送到服务器。

第一批埋点事件和崩溃报告会在稍后上报到服务器并得到处理。报告可能在一段时间后才能在 Flurry 的 Dashboard 中出现。然后你可以进入 Dashboard 查看这些事件和崩溃报告。你应该会看到与下面截图类似的内容，以下截图是为我的应用从 Dashboard 中截取的。

图 1-5 展示了用户会话，用户会话指的是有多少用户在一天中至少启动了应用一次。多次运行可能被认定是同一个会话，也可能不会被认定是同一个会话，这主要取决于多次运行之间的时间间隔。

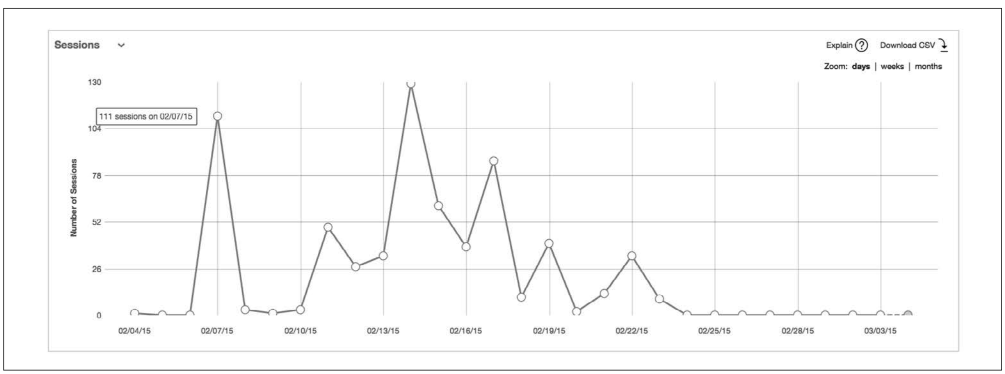
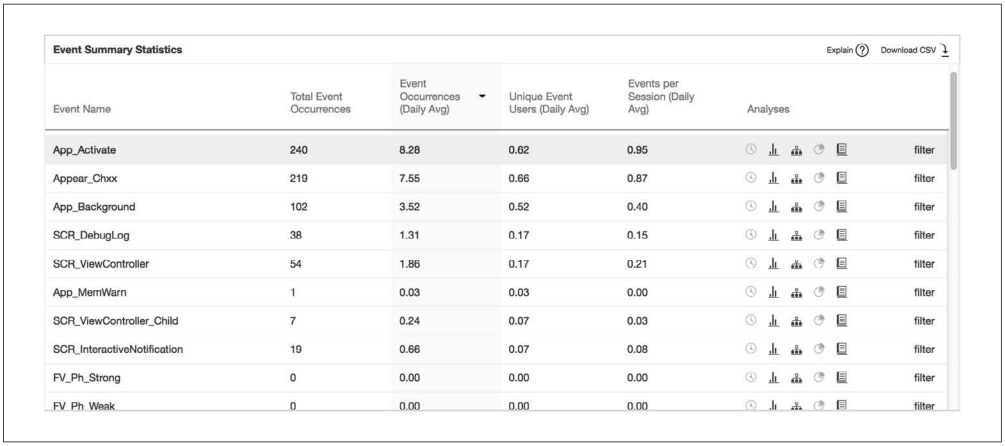


图 1-5：用户会话报告

图 1-6 展示了每个埋点事件的详细崩溃情况。这份报告更加有用，因为它可以让我们深入了解应用的使用率。比如，它精确地指出了应用中的哪些部分比其他部分使用频度更高。

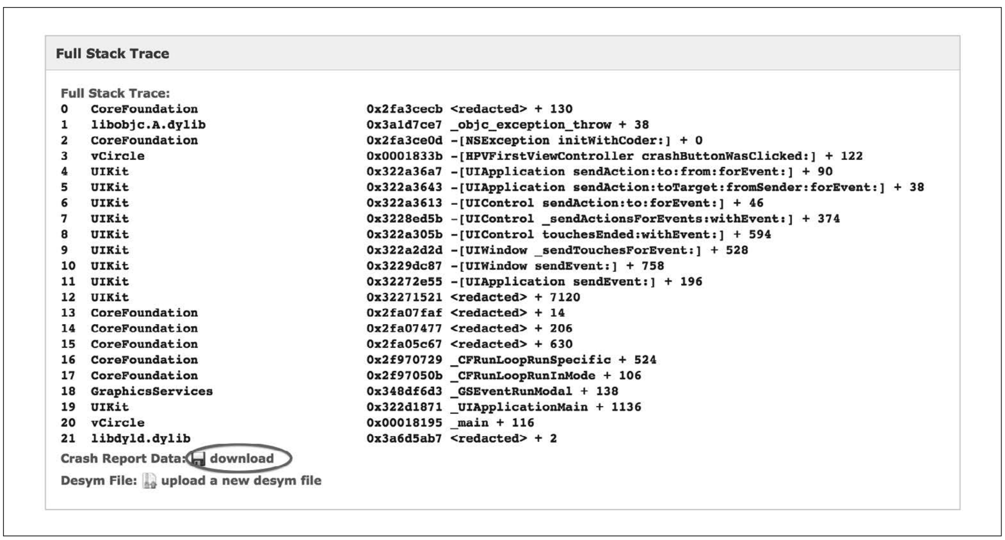


Event Summary Statistics — Explain — Download CSV

Event Name	Total Event Occurrences	Event Occurrences (Daily Avg)	Unique Event Users (Daily Avg)	Events per Session (Daily Avg)	Analyses
App_Activate	240	8.28	0.62	0.95	filter
Appear_Chxx	219	7.55	0.66	0.87	filter
App_Background	102	3.52	0.52	0.40	filter
SCR_DebugLog	38	1.31	0.17	0.15	filter
SCR_ViewController	54	1.86	0.17	0.21	filter
App_MemWarn	1	0.03	0.03	0.00	filter
SCR_ViewController_Child	7	0.24	0.07	0.03	filter
SCR_InteractiveNotification	19	0.66	0.07	0.08	filter
FV_Ph_Strong	0	0.00	0.00	0.00	filter
FV_Ph_Weak	0	0.00	0.00	0.00	filter

图 1-6：事件——更重要的报告

如果查看图 1-7 中的崩溃报告，你会注意到其中的 download 链接，这个链接可以下载崩溃日志。点击链接，下载日志。看着很熟悉是吗？



Full Stack Trace

```
Full Stack Trace:
0  CoreFoundation        0x2fa3cecb <redacted> + 130
1  libobjc.A.dylib       0x3a1d7ce7 _objc_exception_throw + 38
2  CoreFoundation        0x2fa3ce0d -[NSException initWithCoder:] + 0
3  vCircle               0x0001833b -[HPVFirstViewController crashButtonWasClicked:] + 122
4  UIKit                 0x322a36a7 -[UIApplication sendAction:to:from:forEvent:] + 90
5  UIKit                 0x322a3643 -[UIApplication sendAction:toTarget:fromSender:forEvent:] + 38
6  UIKit                 0x322a3613 -[UIControl sendAction:to:forEvent:] + 46
7  UIKit                 0x3228ed5b -[UIControl _sendActionsForEvents:withEvent:] + 374
8  UIKit                 0x322a305b -[UIControl touchesEnded:withEvent:] + 594
9  UIKit                 0x322a2d2d -[UIWindow _sendTouchesForEvent:] + 528
10 UIKit                 0x3229dc87 -[UIWindow sendEvent:] + 758
11 UIKit                 0x32272e55 -[UIApplication sendEvent:] + 196
12 UIKit                 0x32271521 <redacted> + 7120
13 CoreFoundation        0x2fa07faf <redacted> + 14
14 CoreFoundation        0x2fa07477 <redacted> + 206
15 CoreFoundation        0x2fa05c67 <redacted> + 630
16 CoreFoundation        0x2f970729 _CFRunLoopRunSpecific + 524
17 CoreFoundation        0x2f97050b _CFRunLoopRunInMode + 106
18 GraphicsServices      0x348df6d3 _GSEventRunModal + 138
19 UIKit                 0x322d1871 _UIApplicationMain + 1136
20 vCircle               0x00018195 _main + 116
21 libdyld.dylib         0x3a6d5ab7 <redacted> + 2
Crash Report Data: download
Desym File: upload a new desym file
```

图 1-7：崩溃报告——最重要的报告

在 iTunes Connect 中查看崩溃报告

Apple 提供了下载崩溃报告的服务。你可以利用该服务下载 TestFlight 或 App Store 所发布的应用的最近版本，也可以构建相应的崩溃报告。理论上来说，有了这个服务就不再需要第三方的崩溃报告工具了。

但有一个棘手的问题。除非用户同意与开发人员分享崩溃数据，否则崩溃日志不会发送至 Apple。TestFlight 用户自动同意了分享崩溃数据。但产品应用（通过 App Store 分发的应用）必须由用户开启分享。

要想实现这一点，用户需要进入设置应用，打开“隐私→诊断与用量”（Privacy → Diagnostics & Usage），然后选择“自动发送”（Automatically Send）选项（见图 1-8）。

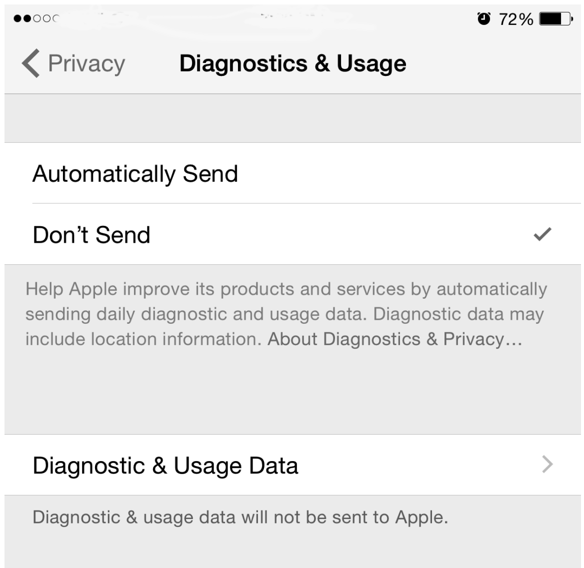


图 1-8：在设备中设置发送崩溃报告

这里存在两个问题。首先，用户不能在你的应用内设置这个选项，他们需要进入设置应用并进入特定的设置项。第二点更为重要，这项设置会对所有的应用生效：用户无法只为特定的应用发送崩溃报告。

使用第三方的崩溃报告工具可以确保你能够控制整体体验和用户设置，以便向服务器发送崩溃报告。

1.4.4 日志

日志是无价之宝，可以用于了解应用发生了什么事。

日志和埋点之间存在着细微的差别。埋点可以看作日志的子集。被埋点的任何数据都应该记录在日志中。

埋点承担了为聚合分析发布关键性能数据的职责，日志则提供了用于在不同级别跟踪应用的细节信息，比如 `debug`、`Verbose`、`info`、`warning` 和 `Error`。日志的记录会贯穿应用的整个生命周期，而埋点只应该用在开发的特定阶段。

埋点数据会发送到服务器，日志是记录在设备本地。

就日志而言，我们可以通过 CocoaPods 引入 `CocoaLumberjack` 来使用。

例 1-5 展示了添加到 Podfile 的一行代码，以便引入库。完成相关改动后，运行 `pod update` 以更新 Xode 的工作空间。

例 1-5　为 `CocoaLumberjack` 配置 Podfile

```
pod 'CocoaLumberjack', '~> 2.0'
```

`CocoaLumberjack` 是一个扩展性很强的框架，捆绑了一系列内置的日志记录器，这些记录器可以向不同的目标发送信息。例如，使用 `DDASLLogger` 可以向 Apple System Log（ASL，`NSLog` 方法的默认位置）记录日志。类似地，使用 `DDFileLogger` 可以向文件记录日志。可以在应用运行期间配置记录器。

`DDLog<Level>` 宏指令可以用于记录某个特定层级的日志。层级越高，信息越重要。最高级别是 `Error`，最低级别是 `Verbose`。实际记录消息的最低层级可以配置在每个文件层级、每个 Xcode 配置层级、每个日志器层级或全局。

以下的宏指令可供使用。

- `DDLogError`
  表示不可恢复的错误。
- `DDLogWarn`
  表示可恢复的错误。
- `DDLogInfo`
  表示非错误的信息。
- `DDLogDebug`
  表示数据主要用于调试。
- `DDLogVerbose`
  几乎提供了所有的细节，主要用于跟踪执行过程中的控制流。

这些宏指令有着与 `NSLog` 相同的签名。这意味着你可以直接用适合的 `DDLog<Level>` 调用来取代 `NSLog`。

例 1-6 展示了配置和使用这个库的代表性代码。

例 1-6　配置和使用 `CocoaLumberjack`

```
//设置
-(void)setupLogger { ❶

#if _DEBUG
    [DDLog addLogger:[DDASLLogger sharedInstance]]; ❷
#endif

    DDFileLogger fileLogger = [[DDFileLogger alloc] init]; ❸
    fileLogger.rollingFrequency = 60 * 60 * 24;
```

```
    fileLogger.logFileManager.maximumNumberOfLogFiles = 7;

    [DDLog addLogger:fileLogger]; ❹
}

//在一些文件中使用记录器

#if _DEBUG ❺
    static const DDLogLevel ddLogLevel = DDLogLevelVerbose;
#elsif MY_INTERNAL_RELEASE
    static const DDLogLevel ddLogLevel = DDLogLevelDebug;
#else
    static const DDLogLevel ddLogLevel = DDLogLevelWarn;
#end

-(void)someMethod {
  DDVerbose(@"someMethod has started execution"); ❻
  //...
  DDError(@"Ouch! Error state. Don't know what to do");
  //...
  DDVerbose(@"someMethod has reached its end state");
}
```

❶ 最有可能在 `application:didFinishLaunchingWithOptions:` 调用这个方法。

❷ 当连接到 Xcode 时，只在调试模式下将日志记录到 ASL。你不会希望这类日志在产品环境的设备中记录下来。

❸ 文件日志记录器，配置为每 24 小时（`rollingFrequence`）创建一个新文件，同时最多允许创建 7 个文件（`maximumNumberOfLogFiles`）。

❹ 注册日志记录器。

❺ 将日志的级别（`ddLogLevel`）设置为合适的值。这里我们可以这样设置：开发阶段输出最多的细节；内部测试阶段（`MY_INTERNAL_RELEASE` 是一个自定义的标记）输出少量细节信息（`debug level`）；面向终端用户的分发包只输出错误信息。

❻ 记录一些信息。在 `DDLogLevelVerbose` 级别中，所有信息都会被记录；在 `DDLogLevelWarn` 级别中，只有错误信息会被记录。

> 建议在应用委托的 `application:didFinishLaunchingWithOptions:` 回调中调用此方法。

1.5 小结

我们在本章了解了影响应用性能的因素。性能不仅涉及用户体验，更关系到应用能否高效运行。

我们查看了一些影响应用性能的关键属性。在包含测量和追踪的各项指标中，这些属性成为了性能的关键指示器。

我们讨论了性能分析的概念，并宽泛地介绍了两类分析技术：采样与埋点。还介绍了一些代码改动，以便对应用进行埋点。然后我们通过使用被埋点的应用来触发事件。

最后，我们在类中添加一些样板代码以帮助进行埋点和记录日志。

第二部分的章节主要关注定义性能的每个属性。每章都从定义和评审属性开始，然后讨论一些潜在的问题，并用实际的代码来解决这些问题。

第二部分

核心优化

我们即将探讨最核心的优化，以便能够使用 Objective-C 编写高效的应用。这些优化构成了每个应用的基础，它们无处不在。这些优化和具体选用的 API 无关，和在应用的哪一层实现无关，甚至和应用的目标也没什么关系，因为它们在整个应用领域都适用。

我们要讨论的优化包括以下方面：

- 内存管理
- 能耗
- 并发编程

第 2 章

内存管理

iPhone 和 iPad 设备的内存资源非常有限。如果某个应用的内存使用量超过了单个进程的上限，那么它就会被操作系统终止使用。[1] 正是由于这个原因，成功的内存管理在 iOS 应用的实现过程中扮演着核心的角色。

苹果公司在 2011 年的全球开发者大会上指出，90% 的应用崩溃与内存管理有关。其中最主要的原因是错误的内存访问和保留环所引起的内存泄漏。

与（基于垃圾回收的）Java 运行时不同，Objective-C 和 Swift 的 iOS 运行时使用引用计数。使用引用计数的负面影响在于，如果开发人员不够小心，那么可能会出现重复的内存释放和循环引用的情况。

因此，理解 iOS 的内存管理是十分重要的。

我们将在本章学习以下知识点：

- 内存消耗（例如，应用如何消耗内存）
- 内存管理模型（例如，iOS 运行时如何管理内存）
- 语言架构——我们将介绍 Objective-C 的架构及一些实用特性
- 在不影响用户体验的前提下，采用减少内存使用的最佳实践

2.1 内存消耗

内存消耗指的是应用消耗的 RAM。

注 1：iOS Developer Library, “Technical Note TN2151: Understanding and Analyzing iOS Application CrashReports (https://developer.apple.com/library/ios/technotes/tn2151/_index.html#//apple_ref/doc/uid/DTS40008184-CH1-UNDERSTANDING_LOW_MEMORY_REPORTS).

iOS 的虚拟内存模型并不包含交换内存，与桌面应用不同，这意味着磁盘不会被用来分页内存。最终的结果是应用只能使用有限的 RAM。这些 RAM 的使用者不仅包括在前台运行的应用，还包括操作系统服务，甚至还包括其他应用所执行的后台任务。

应用中的内存消耗分为两部分：栈大小和堆大小。接下来的两节将逐一介绍这两部分。

2.1.1 栈大小

应用中新创建的每个线程都有专用的栈空间，该空间由保留的内存和初始提交的内存组成。栈可以在线程存在期间自由使用。线程的最大栈空间很小，这就决定了以下的限制。

- 可被递归调用的最大方法数
  每个方法都有其自己的栈帧，并会消耗整体的栈空间。例如，如例 2-1 所示，如果你调用 `main`，那么 `main` 将调用 `method1`，而 `method1` 又将调用 `method2`，这就存在三个栈帧了，且每个栈帧都会消耗一定字节的内存。图 2-1 展示了线程栈随时间的变化。

例 2-1 调用树

```
main() {
    method1();
}

method1() {
    method2();
}
```

- 一个方法中最多可以使用的变量个数
  所有的变量都会载入方法的栈帧中，并消耗一定的栈空间。
- 视图层级中可以嵌入的最大视图深度
  渲染复合视图将在整个视图层级树中递归地调用 `layoutSubViews` 和 `drawRect` 方法。如果层级过深，可能会导致栈溢出。

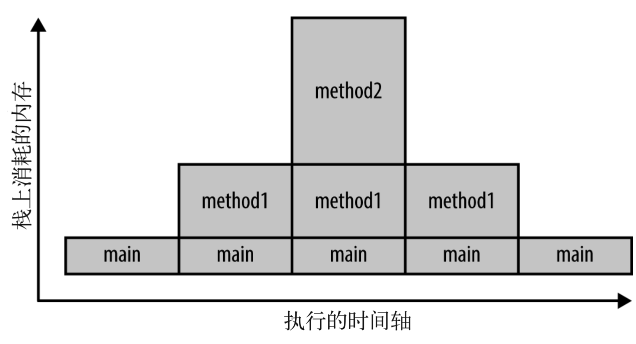


图 2-1：包含每个方法的栈框架的栈

2.1.2 堆大小

每个进程的所有线程共享同一个堆。一个应用可以使用的堆大小通常远远小于设备的 RAM 值。例如，iPhone 5S 拥有大约 1GB 的 RAM，但分配给一个应用的堆大小最多不到 512MB。应用并不能控制分配给它的堆。只有操作系统才能管理堆。[2]

使用 `NSString`、载入图片、创建或使用 JSON/XML 数据、使用视图等都会消耗大量的堆内存。如果你的应用大量使用图片（与 Flickr 和 Instagram 应用类似），那么你需要格外关注平均值和峰值内存使用的最小化。

图 2-2 展示了可能出现在一个应用某个时刻的一个典型堆。

在图 2-2 中，由 `main` 方法启动的主线程创建了 `UIApplication`。我们假设某个时间点的窗体包含了一个 `UITableView`，当必须渲染表格中的一行时，`UITableView` 调用了 `UITableViewDataSource` 的 `tableView:cellForRowAtIndex:` 方法。

通过名为 `photos` 的 `NSArray` 属性，数据源引用了全部的照片。如果处理不够谨慎，这个数组将会非常大，从而导致很高的峰值内存使用。解决方案之一是在数组中存储固定数量的图片，并在用户滚动视图时换入或换出图片。这个固定的数值将决定此应用的平均内存使用。

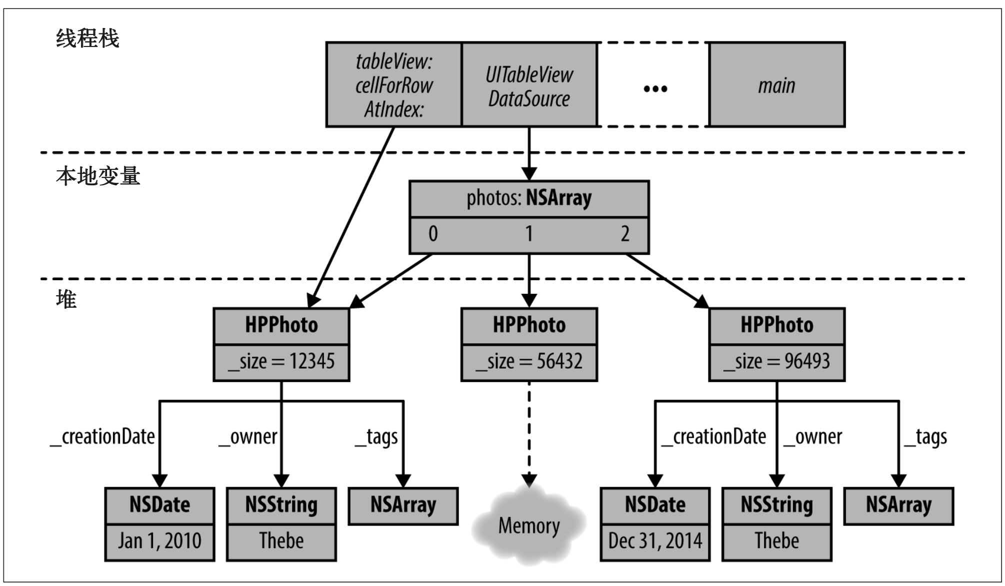


图 2-2：在 `UITableViewDataSource` 中展示 `HPPhoto` 模型使用情况的堆

数组中的每一项都是 `HPPhoto` 类型，代表了一张照片。`HPPhoto` 储存了与对象有关的数据，如照片的尺寸、创建日期、拥有者信息、标签、与照片关联的网络 URL（图中没有展示）、对本地缓存的引用（图中没有展示），等等。

注 2：Stack Overflow,“iOS Equivalent to Increasing Heap Size”(http://stackoverflow.com/a/25369670).

与通过类创建的对象相关的所有数据都存放在堆中。

类可能包含属性或值类型的实例变量（iVars），如 int、char 或 struct。但因为对象是在堆内创建的，所以它们只消耗堆内存。

当对象被创建并被赋值时，数据可能会从栈复制到堆。类似地，当值仅在方法内部使用时，它们也可能会被从堆复制到栈。这可能是个代价昂贵的操作。例 2-2 重点展示了从栈复制到堆以及从堆复制到栈的情况。

例 2-2　堆与栈

```
@interface AClass ❶

@property (nonatomic, assign) NSInteger anInteger; ❷
@property (nonatomic, copy) NSString *aString; ❸

@end

//一些其他的类
-(AClass *) createAClassWithInteger:(NSInteger)i
    string:(NSString *)s { ❹

    AClass *result = [AClass new];
    result.anInteger = i; ❺
    result.aString = s; ❻
}

-(void) someMethod:(NSArray *)items { ❼
    NSInteger total = 0;
    NSMutableString *finalValue = [NSMutableString string];

    for(AClass *obj in items) {
        total += obj.anInteger; ❽
        [finalValue appendString:obj.aString]; ❾
    }
}
```

❶ 类 AClass 包含两个属性。

❷ anInteger 为 NSInteger 类型，通过传值方式进行传递。

❸ aString 为 NSString * 类型，通过引用传递。

❹ createAClassWithInteger:string: 方法（存在于其他某个不相关的类）初始化了 AClass。这个方法拥有创建对象时需要的值。

❺ i 的值在栈上。但赋值给属性时，它必须被复制到堆中，因为那是存储 result 的地方。

❻ 虽然 NSString * 通过引用传递，但这个属性被标记为 copy。这意味着它的值必须被复制或克隆，这取决于 [-NSCopying copyWithZone:] 方法的实现。

❼ someMethod: 方法对 AClass 对象的数组进行操作。

❽ 使用 anInteger 时，它的值必须先复制到栈然后才能进行进一步的处理。在本示例中，它的值加到 total。

❾ aString 在使用时通过引用传递。在本示例中，appendString: 使用了 aString 对象的引用。

保持应用的内存需求总是处于 RAM 的较低占比是一个非常好的主意。虽然没有强制规定，但强烈建议使用量不要超过 80%~85%，要给操作系统的核心服务留下足够多的内存。

不要忽视 `didReceiveMemoryWarning` 信号。

2.2 内存管理模型

我们将在本节中学习 iOS 的运行时是如何管理内存的，以及它对代码的影响。

内存管理模型基于*持有关系*的概念。如果一个对象正处于*被持有*状态，那它占用的内存就不能被回收。

当一个对象创建于某个方法的内部时，那该方法就*持有*这个对象了。如果这个对象从方法返回，则调用者声称建立了*持有关系*。这个值可以赋值[3]给其他变量，对应的变量同样会声称建立了持有关系。

一旦与某个对象相关的任务全部完成，那么就是*放弃*了持有关系。这一过程没有转移持有关系，而是分别增加或减少了持有者的数量。当持有者的数量降为零时，对象会被释放（https://developer.apple.com/library/mac/documentation/Cocoa/reference/Foundation/Classes/nsobject_Class/Reference/Reference.html#//apple_ref/occ/instm/NSObject/dealloc），相关的内存会被回收。

这种持有关系计数通常被正式称为*引用计数*。当你亲自管理时，它被称为*手动引用计数*（manual reference counting，MRC）。虽然现在已经十分罕见，但 MRC 对理解问题很有帮助。现如今的应用大都使用*自动引用计数*（automatic reference counting ，ARC），我们将在 2.5 节中对其进行讨论。

例 2-3 演示了基于引用计数的手动内存管理的基本结构。

例 2-3　通过手动内存管理进行引用计数

```
NSString *message = @"Objective-C is a verbose yet awesome language"; ❶
NSString *messageRetained = [message retain]; ❷
[messageRetained release]; ❸
[message release]; ❹
NSLog(@"Value of message: %@", message); ❺
```

❶ 创建对象、`message` 建立了持有关系，引用计数为 1。

❷ `messageRetained` 建立了持有关系，引用计数增加为 2。

❸ `messageRetained` 放弃了持有关系，引用计数降为 1。

❹ `message` 放弃了持有关系，引用计数降为 0。

❺ 严格来讲，此时 `message` 的值是未定义的。你仍然能像之前那样得到相同的值，因为它对应的内存还没有被回收或重置。

例 2-4 演示了方法是如何对引用计数产生影响的。

注 3：术语“赋值”在此处使用的并不精确，我们将在后文做详细阐述。

例 2-4　方法中的引用计数

```
//一个Person类的部分
-(NSString *) address {
    NSString *result = [[NSString alloc]
            initWithFormat:@"%@\n%@\n%@, %@",
            self.line1, self.line2, self.city, self.state]; ❶
    return result;
}

-(void) showPerson:(Person *) p {
    NSString *paddress = [p address]; ❷

    NSLog(@"Person's Address: %@", paddress);

    [paddress release]; ❸
}
```

❶ 首次创建对象，`result` 指向内存的引用计数为 1。

❷ 通过 `paddress`（指向 `result`）指向的内存的引用计数仍然是 1。`showPerson:` 方法通过 `address` 按钮创建了对象，是对象的持有者。对象不应该被再次持有（`retain`）。

❸ 放弃持有关系；引用计数降为 0。

在例 2-4 中，`showPerson:` 并不知道 `address` 方法是创建了一个新的对象还是重用了旧的对象。但它知道引用计数加 1 后，对象会被返回。因此，这里没有继续持有 `address`。一旦完成任务，它将释放对象。如果对象的引用计数是 1，那么它将变成 0，并且被回收。

苹果公司和 LLVM 的官方文档更喜欢使用术语*持有关系*。本书将交替使用*持有关系*和*引用计数*这两个术语。

2.3　自动释放对象

自动释放对象让你能够放弃对一个对象的持有关系，但延后对它的销毁。当在方法中创建一个对象并需要将其返回时，自动释放就显得非常有用。自动释放可以帮助在 MRC 中管理对象的生命周期。

以严格的 Objective-C 命名规范为标准，在例 2-4 中，没什么能表示 `address` 方法持有了返回的字符串。因此，方法的调用者 `showPerson:` 也不应该释放返回的字符串，这可能会导致发生内存泄漏。加入 `[paddress release]` 这行代码的目的是为了指明这种情况。

那么，正确使用 `address` 方法的代码是什么样的呢？

以下是两种可能的解决方案。

- 不要使用 `alloc` 或相关的方法。
- 对返回的对象使用延时释放。

使用 `NSString` 时，第一个修复版本很容易实现。更新后的代码见例 2-5。

例 2-5　方法中引用计数的修复代码

```
-(NSString *) address {
    NSString *result = [NSString
            stringWithFormat:@"%@\n%@\n%@, %@",
            self.line1, self.line2, self.city, self.state]; ❶
    return result;
}

-(void) showPerson:(Person *) p {
    NSString *paddress = [p address];

    NSLog(@"Person's Address: %@", paddress);
    ❷
}
```

❶ 不要使用 `alloc` 方法。

❷ 由于 `showPerson:` 方法没有创建实体对象，因此不要在 `showPerson:` 方法中使用 `release` 方法。

然而，这种修复方法在没有使用 `NSString` 的情况下并不容易实现，因为通常很难找到能够满足需要的适合方法。例如，当使用第三方类库或者某个类有多个用于创建对象的方法时，到底是哪个方法保持了持有关系并不明确。

延迟销毁大显神威的机会来了。

`NSObject` 协议定义了可被用于延迟释放的 `autorelease` 消息。可在从方法中返回对象时使用它。

例 2-6 显示了使用 `autorelease` 的更新代码。

例 2-6　使用 `autorelease` 的引用计数

```
-(NSString *) address
{
    NSString *result = [[[NSString alloc]
            initWithFormat:@"%@\n%@\n%@, %@",
            self.line1, self.line2, self.city, self.state]
        autorelease];
    return result;
}
```

可以用以下规则来分析代码。

(1) 持有的对象（在上述示例中是 `NSString`）是 `alloc` 方法返回的。

(2) 确保没有内存泄漏，你必须在失去引用之前放弃持有关系。

(3) 但是，如果使用了 `release`，那么对象的释放将发生在返回之前，因而方法将返回一个无效的引用。

(4) `autorelease` 表明你想要放弃持有关系，同时允许方法的调用者在对象被释放之前使用对象。

当创建一个对象并将其从非 `alloc` 方法返回时，应使用 `autorelease`。这样可以确保对象将被释放，并尽量在调用方法执行完成时立即释放。

2.4　自动释放池块

自动释放池块是允许你放弃对一个对象的持有关系、但可避免它立即被回收的一个工具。当从方法返回对象时，这种功能非常有用。

它还能确保在块内创建的对象会在块完成时被回收。这在创建了多个对象的场景中非常有用。本地的块可以用来尽早地释放其中的对象，从而使内存用量保持在较低的水平。

自动释放池块用 `@autoreleasepool` 表示。

打开示例工程的 main.m 文件，你会发现例 2-7 中的代码。

例 2-7　main.m 中的 `@autoreleasepool` 块

```
int main(int argc, char * argv[]) {
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil,
            NSStringFromClass([HPAppDelegate class]));
        }
}
```

块中收到过 `autorelease` 消息的所有对象都会在 `autoreleasepool` 块结束时收到 `release` 消息。更加重要的是，每个 `autorelease` 调用都会发送一个 `release` 消息。这意味着如果一个对象收到了不止一次的 `autorelease` 消息，那它也会多次收到 `release` 消息。这一点很棒，因为这能保证对象的引用计数下降到使用 `autoreleasepool` 块之前的值。如果计数为 0，则对象将被回收，从而保持较低的内存使用率。

看了 `main` 方法的代码后，你会发现整个应用都在一个 `autoreleasepool` 块中，这意味着所有的 `autorelease` 对象最后都会被回收，不会导致内存泄漏。

与其他的代码块一样，`autoreleasepool` 块可以被嵌套，如例 2-8 所示。

例 2-8　嵌套的 `autoreleasepool` 块

```
@autoreleasepool {
 // 一些代码
 @autoreleasepool {
  // 更多代码
        }
}
```

因为从一个方法进入另一个方法会传递控制权，所以在同一方法内部使用嵌套的 `autoreleasepool` 块并不常见。但是，被调用的方法也可能拥有自己的 `autoreleasepool` 块，以提前执行对象的回收。

自动释放池块无处不在

Cocoa 框架希望代码能在 autoreleasepool 块内执行，否则 autorelease 对象将无法被释放，从而导致应用发生内存泄漏。

AppKit 和 UIKit 框架将事件 – 循环的迭代放入了 autoreleasepool 块中。因此，通常不需要你自己再创建 autoreleasepool 块了。

但在一些特定情况下，你很可能想创建自己的 autoreleasepool 块，例如以下这些情况。

- 当你有一个创建了很多临时对象的循环时
  在循环中使用 autoreleasepool 块可以为每个迭代释放内存。虽然迭代前后最终的内存使用相同，但你的应用的最大内存需求可以大大降低。

  例 2-9 提供了一些例子，其中包括使用 autoreleasepool 时编写的好与不好的代码实现。

- 当你创建一个线程时

  每个线程都将有它自己的 autoreleasepool 块栈。主线程用自己的 autoreleasepool 启动，因为它来自统一生成的代码。然而，对于任何自定义的线程，你必须创建自己的 autoreleasepool。

  你可以通过例 2-10 查看示例代码。

例 2-9　循环中的自动释放池块

```
//不良代码 ❶
{
    @autoreleasepool {
        NSUInteger *userCount = userDatabase.userCount;

        for(NSUInteger *i = 0; i < userCount; i++) {
            Person *p = [userDatabase userAtIndex:i];

            NSString *fname = p.fname;
            if(fname == nil) {
                fname = [self askUserForFirstName];
            }

            NSString *lname = p.lname;
            if(lname == nil) {
                lname = [self askUserForLastName];
            }
            //...
            [userDatabase updateUser:p];
        }
    }
}

//好的代码 ❷
{
    @autoreleasepool {
```

```
            NSUInteger *userCount = userDatabase.userCount;

            for(NSUInteger *i = 0; i < userCount; i++) {

                @autoreleasepool {
                    Person *p = [userDatabase userAtIndex:i];

                    NSString *fname = p.fname;
                    if(fname == nil) {
                        fname = [self askUserForFirstName];
                    }

                    NSString *lname = p.lname;
                    if(lname == nil) {
                        lname = [self askUserForLastName];
                    }
                    //...
                    [userDatabase updateUser:p];
                }
            }
        }
    }
```

❶ 这段代码很糟糕，因为只有一个 `autoreleasepool`，而且内存清理工作要在所有的循环迭代完成之后才能进行。

❷ 这个示例中有两个 `autoreleasepool`，内层的 `autoreleasepool` 确保在每次循环迭代完成后清理内存，从而导致更少的内存需求。

例 2-10　自定义线程中的自动释放池块

```
-(void)myThreadStart:(id)obj {
    @autoreleasepool {
        //新线程的代码
    }
}

//其他地方
{
    NSThread *myThread = [[NSThread alloc] initWithTarget:self
        selector:@selector(myThreadStart:)
        object:nil];

    [myThread start];
}
```

2.5　自动引用计数

持续跟踪 `retain`、`release` 和 `autorelease` 并不容易。要想找出是谁在什么时间和地点向谁发送了这些消息就更难了。

苹果公司在 2011 年的全球开发者大会上介绍了解决这一问题的方案——ARC。iOS 应用的新兴语言 Swift 同样也在使用 ARC。与 Objective-C 不同的是，Swift 不支持 MRC。

ARC 是一种编译器特性[4]。它评估了对象在代码中的生命周期，并在编译时自动注入适合的内存管理调用。编译器还会生成适合的 `dealloc` 方法。这意味着与跟踪内存使用（如确保对象被及时回收了）有关的最大难题被解决了。

图 2-3 演示了使用 MRC 与 ARC 的开发时间对比。由于代码减少，使用 ARC 开发的进程会大大加快。

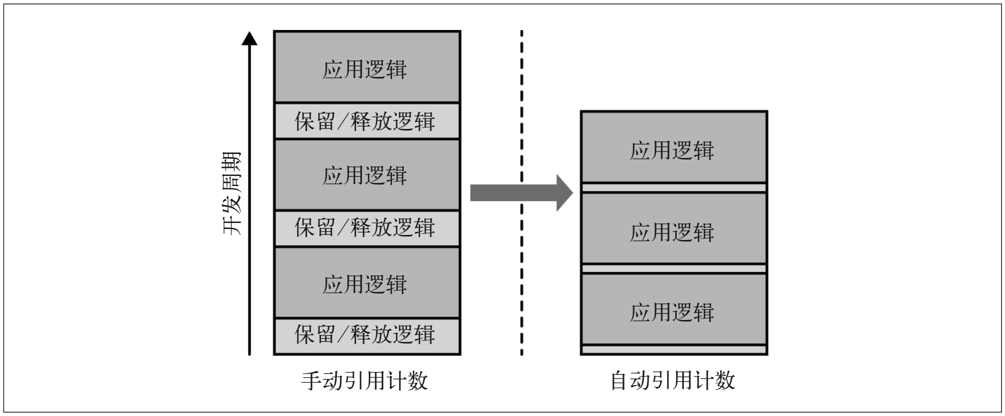


图 2-3：ARC 减少了开发时间，减轻了负担

你需要确保在 Xcode 工程设置中开启了 ARC，这从 Xcode 5 开始成为一项默认设置（见图 2-4）。

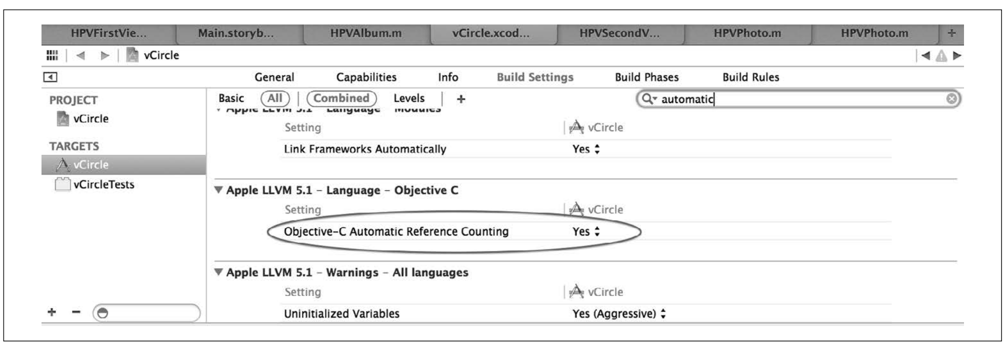


图 2-4：ARC 在 Xcode 中的项目设定

注 4：你可以从 LLVM 的站点找到自动引用计数的完整规范（http://clang.llvm.org/docs/AutomaticReference-Counting.html）。

没有使用 ARC 的依赖项

现阶段不支持 ARC 或没有相关解决方案的依赖项目已经非常少了。

但如果真的遇到了这样的依赖项目，那么你需要对一个或多个文件禁用 ARC。

要想禁用 ARC，需进入 Targets->Build Phases->Compile Sources，选择必须禁用 ARC 的文件，然后添加编译器标记 `-fno-objc-arc`，如图 2-5 所示。

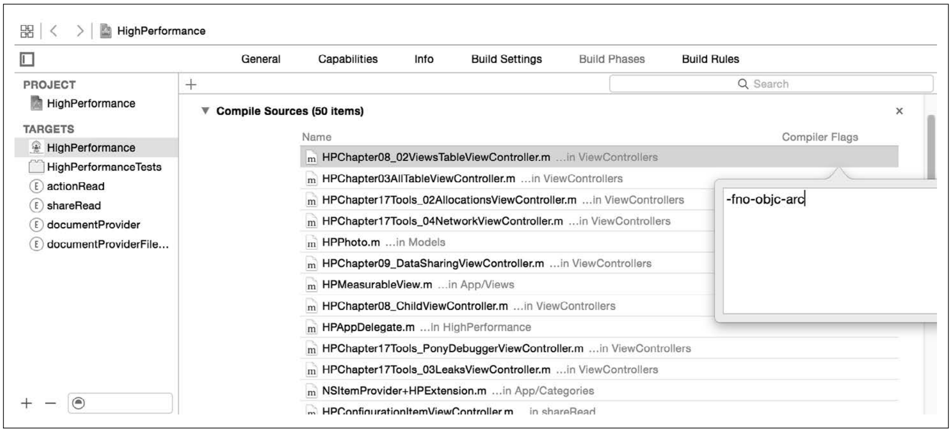


图 2-5：在文件级禁用 ARC

相同的选项可用于创建混合模式的类，这个类可以包含基于 MRC 的 Category。代码写在 Category 的文件中，对应的文件可以禁用 ARC。

ARC的规则

ARC 强制推行了编写代码时需要遵循的一些规则。这些规则的意图是提供一个明确的内存管理模型。在某些情况下，这些规则的目的是强制实施最佳实践；在其他情况下，它们可以简化代码，直接的必然结果就是开发人员不必直接进行内存管理。[5] 这些规则由编译器强制执行，会导致编译期时的错误而不是运行时的崩溃。以下就是编译器的 ARC 规则。

- 不能实现或调用 `retain`、`release`、`autorelease` 或 `retainCount` 方法。这一限制不仅针对对象，对选择器同样有效。因此，`[obj release]` 或 `@selector(retain)` 是编译时的错误。
- 可以实现 `dealloc` 方法，但不能调用它们。不仅不能调用其他对象的 `dealloc` 方法，也不能调用超类。`[super dealloc]` 是编译时的错误。
  但你仍然可以对 Core Foundation 类型的对象调用 `CFRetain`、`CFRelease` 等相关方法。
- 不能调用 `NSAllocateObject` 和 `NSDeallocateObject` 方法。应使用 `alloc` 方法创建对象，运行时负责回收对象。

注 5：iOS Developer Library,“Transitioning to ARC Release Notes” (http://apple.co/1KfifW3](http://apple.co/1KfifW3).

- 不能在 C 语言的结构体内使用对象指针。
- 不能在 id 类型和 void * 类型之间自动转换。如果需要，那么你必须做显示转换。
- 不能使用 NSAutoreleasePool，要替换使用 autoreleasepool 块。
- 不能使用 NSZone 内存区域。
- 属性的访问器名称不能以 new 开头，以确保与 MRC 的互操作性。例 2-11 演示了这一点。
- 虽然总的来说需要避免许多事情，但仍然可以混合使用 ARC 和 MRC 代码（我们在本节前面讨论过了这个问题）。

例 2-11　开启了 ARC 后的访问器名称

```
//未允许
@property NSString * newTitle;

//允许
@property (getter=getNewTitle) NSString * newTitle;
```

牢记这些规则，我们可以更新例 2-5 中的代码。更新后的代码见例 2-12。

例 2-12　开启 ARC 后被更新的代码

```
-(NSString *) address
{
    NSString *result = [[NSString alloc] initWithFormat:@"%@\n%@\n%@, %@",
        self.line1, self.line2, self.city, self.state]; ❶
    return result;
}

-(void) showPerson:(Person *) p
{
    NSString *paddress = [p address];

    NSLog(@"Person's Address: %@", paddress);
    ❷
}
```

❶ 此处无需调用 autorelease。你不能在 result 对象上调用 autorelease 或 retain 方法。
❷ 你不能再对 paddress 调用 release 方法。

2.6　引用类型

ARC 带来了新的引用类型：*弱引用*。深入理解这些引用类型对内存管理非常重要。支持的类型包括以下两种。

- 强引用
  强引用是默认的引用类型。被强引用指向的内存不会被释放。强引用会对引用计数加 1，从而扩展对象的生命周期。

- 弱引用
  弱引用是一种特殊的引用类型。它不会增加引用计数，因而不会扩展对象的生命周期。在启用了 ARC 的 Objective-C 编程中，弱引用格外重要。

> **其他类型的引用**
>
> Objective-C 当前并不支持其他类型的引用。但了解一下其他类型也是非常有趣的事情。
>
> - 软引用
>   软引用与弱引用非常相似，只是前者没有那么迫切地抛弃它所引用的对象。如果一个对象只有弱引用存在，那么这个对象会在下个垃圾回收周期被回收；如果一个对象只有软引用可达，那么这个对象一般还能再坚持一会。
> - 幽灵引用
>   这是力量最弱的引用类型，会被最早地回收清理。幽灵引用的对象与已回收的对象比较相似，但是前者的内存没有被回收利用。
>
> 这些引用类型没有基于引用计数系统。它们更适合用于垃圾回收系统。

2.6.1　变量限定符

ARC 为变量供了四种生命周期限定符。

- `__strong`
  这是默认的限定符，无需显示引入。只要有强引用指向，对象就会长时间驻留在内存中。可以将 `__strong` 理解为 `retain` 调用的 ARC 版本。
- `__weak`
  这表明引用不会保持被引用对象的存活。当没有强引用指向对象时，弱引用会被置为 `nil`。可将 `__weak` 看作是 `assign` 操作符的 ARC 版本，只是对象被回收时，`__weak` 具有安全性——指针将自动被设置为 `nil`。
- `__unsafe_unretained`
  与 `__weak` 类似，只是当没有强引用指向对象时，`__unsafe_unretained` 不会被置为 `nil`。可将其看作 `assign` 操作符的 ARC 版本。
- `__autoreleasing`
  `__autoreleasing` 用于由引用使用 `id *` 传递的消息参数。它预期了 `autorelease` 方法会在传递参数的方法中被调用。

使用这些限定符的语义如下：

```
TypeName * qualifier variable;
```

例 2-13 的代码展示了限定符的使用。

例 2-13　使用变量限定符

```
Person * __strong p1 = [[Person alloc] init]; ❶
Person * __weak p2 = [[Person alloc] init]; ❷
Person * __unsafe_unretained p3 = [[Person alloc] init]; ❸
Person * __autoreleasing p4 = [[Person alloc] init]; ❹
```

❶ 创建对象后引用计数为 1，并且对象在 p1 引用期间不会被回收。
❷ 创建对象后引用计数为 0，对象会被立即释放，且 p2 将被设置为 nil。
❸ 创建对象后引用计数为 1，对象会被立即释放，但 p3 不会被设置为 nil。
❹ 创建对象后引用计数为 1，当方法返回时对象会被立即释放。

2.6.2　属性限定符

属性声明有两个新的持有关系限定符：strong 和 weak。此外，assign 限定符的语义也被更新了。一言以蔽之，现在共有六个限定符。

- strong
  默认符，指定了 __strong 关系。
- weak
  指定了 __weak 关系。
- assign
  这不是新的限定符，但其含义发生了改变。在 ARC 之前，assign 是默认的持有关系限定符。在启用 ARC 之后，assign 表示了 __unsafe_unretained 关系。
- copy
  暗指了 __strong 关系。此外，它还暗示了 setter 中的复制语义（https://developer.apple.com/library/mac/documentation/Cocoa/reference/Foundation/Classes/nsobject_Class/Reference/Reference.html#//apple_ref/occ/instm/NSObject/copy）的常规行为。
- retain
  指定了 __strong 关系。
- unsafe_unretained
  指定了 __unsafe_unretained 关系。

例 2-14 展示了这些限定符。 因为 assign 和 unsafe_unretained 只进行值复制而没有任何实质性的检查，所以它们只应该用于值类型（BOOL、NSInteger、NSUInteger，等等）。应避免将它们用于引用类型，尤其是指针类型，如 NSString * 和 UIView *。

例 2-14　使用属性限定符

```
@property (nonatomic, strong) IBOutlet UILabel *titleView;
@property (nonatomic, weak) id<UIApplicationDelegate> appDelegate;
@property (nonatomic, assign) UIView *danglingReference; ❶
@property (nonatomic, assign) BOOL selected; ❷
@property (nonatomic, copy) NSString *name;
@property (nonatomic, retain) HPPhoto *photo; ❸
@property (nonatomic, unsafe_unretained) UIView  *danglingReference;
```

❶ 错误地将 assign 用于指针。
❷ 对值类型正确地使用了 assign 限定符。
❸ retain 是 ARC 纪元之前的老古董，现代的代码已经鲜有使用。在这里添加它只是为了完整性。

2.7 实践环节

现在我们已经学习了用于变量和属性的新的生命周期限定符。让我们亲自实践、更新工程，然后观察效果。

2.7.1 照片模型

创建一个名为 `HPPhoto` 的类，以表示相册中的一张照片。一张照片包括一个 `title`、一个 `url` 和 `comments` 列表。我们也会覆盖 `dealloc` 方法以观察背后的秘密。

先创建一个新的 Objective-C 类：

File → New → iOS → Cocoa Touch → Objective-C class

例 2-15 给出了典型的类声明。

例 2-15　HPPhoto 类

```
//HPPhoto.h
@interface HPPhoto : NSObject

@property (nonatomic, strong) HPAlbum *album;
@property (nonatomic, strong) NSURL *url;
@property (nonatomic, copy) NSString *title;
@property (nonatomic, strong) NSArray *comments;

@end

//HPPhoto.m
@implementation HPPhoto

-(void) dealloc
{
    DDLogVerbose(@"HPPhoto dealloc-ed");
}

@end
```

2.7.2 更新故事板

向故事板中的第一个视图控制器添加一个标签和四个按钮。这些按钮将触发变量的创建，标签用于显示结果。最终呈现的 UI 与图 2-6 所示内容相似。

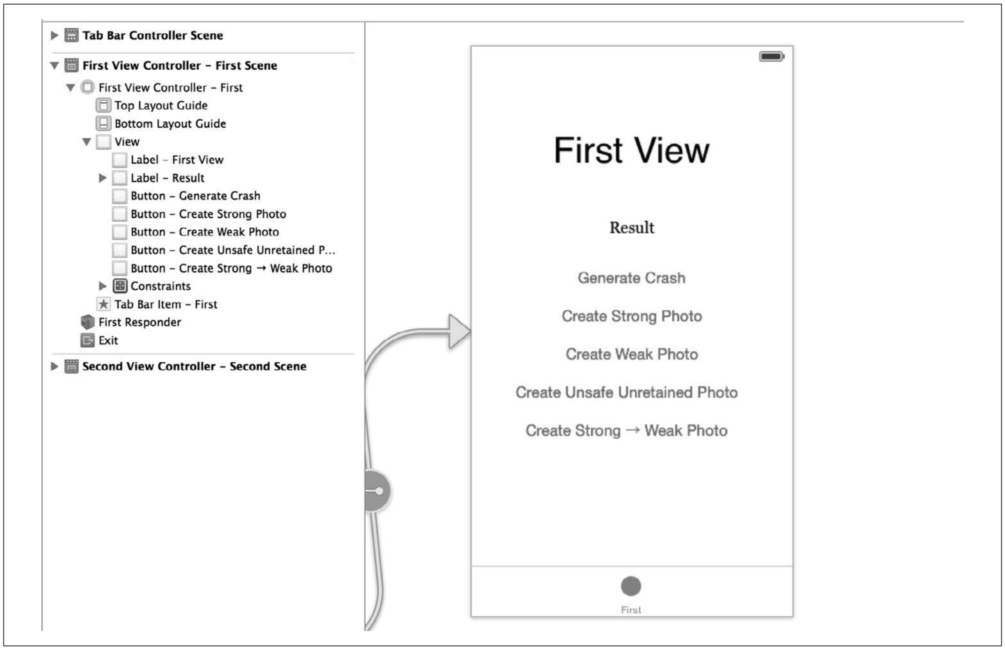


图 2-6：第一个视图控制器的更新视图

我们还在代码中添加了 `IBOutlet` 和 `IBAction` 的引用，如例 2-16 所示。

例 2-16　HPFirstViewController.h 中的引用更新

```
@interface HPFirstViewController : UIViewController

@property (nonatomic, strong) IBOutlet UILabel *resultLabel;

-(IBAction)createStrongPhoto:(id)sender;
-(IBAction)createStrongToWeakPhoto:(id)sender;
-(IBAction)createWeakPhoto:(id)sender;
-(IBAction)createUnsafeUnretainedPhoto:(id)sender;

@end
```

2.7.3　方法实现

我们会在每个方法中做如下事情。

(1) 创建一个 `HPPhoto` 类的实例，并将此实例分配给一个本地引用。
(2) 设置照片的 `title`。
(3) 在 `resultLabel` 中显示该引用是否为 `nil`。如果不是 `nil`，则显示 `title`。

接下来我们将查看每个方法的代码（例 2-17 到例 2-20）。代码的实现大致相同，唯一的区别在于创建引用的类型。注意，我们将不会创建方法来返回引用。我们将在分配内存和创建引用的方法内探索引用。我们还尝试用 `NSLog` 来跟踪生命周期事件的顺序。

此外，我们还研究了一个特殊的情况，即将强引用赋值给弱引用，从而观察对象会发生什么变化。

代码的结果会在 2.7.4 节中介绍。

例 2-17　实现 createStrongPhoto 方法

```
-(IBAction)createStrongPhoto:(id)sender
{
    DDLogDebug(@"%s enter", __PRETTY_FUNCTION__);
        HPPhoto * __strong photo = [[HPPhoto alloc] init];
    DDLogDebug(@"Strong Photo: %@", photo);
        photo.title = @"Strong Photo";

        NSMutableString *ms = [[NSMutableString alloc] init];
        [ms appendString:(photo == nil ? @"Photo is nil" : @"Photo is not nil")];
        [ms appendString:@"\n"];
        if(photo != nil) {
                [ms appendString:photo.title];
        }
        self.resultLabel.text = ms;
        DDLogDebug(@"%s exit", __PRETTY_FUNCTION__);
}
```

例 2-18　实现 createWeakPhoto 方法

```
-(IBAction)createWeakPhoto:(id)sender
{
    DDLogDebug(@"%s enter", __PRETTY_FUNCTION__);
        HPPhoto * __weak wphoto = [[HPPhoto alloc] init];
    DDLogDebug(@"Weak Photo: %@", wphoto);
        wphoto.title = @"Weak Photo";

        NSMutableString *ms = [[NSMutableString alloc] init];
        [ms appendString:(wphoto == nil ? @"Photo is nil" : @"Photo is not nil")];
        [ms appendString:@"\n"];
        if(wphoto != nil) {
                [ms appendString:wphoto.title];
        }
        self.resultLabel.text = ms;
        DDLogDebug(@"%s exit", __PRETTY_FUNCTION__);
}
```

例 2-19　实现 createStrongToWeakPhoto 方法

```
-(void)createStrongToWeakPhoto:(id)sender
{
    DDLogDebug(@"%s enter", __PRETTY_FUNCTION__);
        HPPhoto * sphoto = [[HPPhoto alloc] init];
    DDLogDebug(@"Strong Photo: %@", sphoto);
        sphoto.title = @"Strong Photo, Assigned to Weak";

        HPPhoto * __weak wphoto = sphoto;
        DDLogDebug(@"Weak Photo: %@", wphoto);

        NSMutableString *ms = [[NSMutableString alloc] init];
```

```
        [ms appendString:(wphoto == nil ? @"Photo is nil" : @"Photo is not nil")];
        [ms appendString:@"\n"];
        if(wphoto != nil) {
                [ms appendString:wphoto.title];
        }
        self.resultLabel.text = ms;
    DDLogDebug(@"%s exit", __PRETTY_FUNCTION__);
}
```

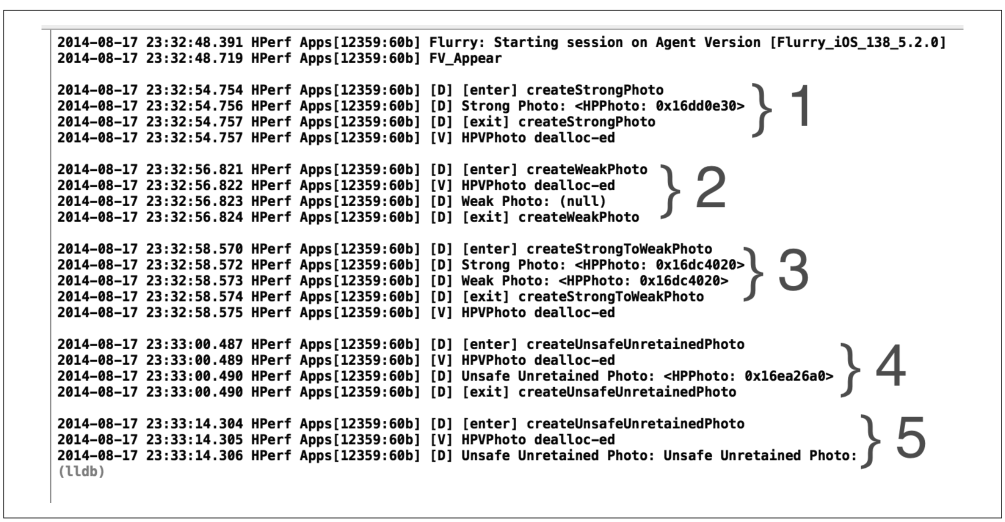
例 2-20　实现 createUnsafeUnretainedPhoto 方法

```
-(void)createUnsafeUnretainedPhoto:(id)sender
{
    DDLogDebug(@"%s enter", __PRETTY_FUNCTION__);
        HPPhoto * __unsafe_unretained wphoto = [[HPPhoto alloc] init];
    DDLogDebug(@"Unsafe Unretained Photo: %@", wphoto);
        wphoto.title = @"Strong Photo";

        NSMutableString *ms = [[NSMutableString alloc] init];
        [ms appendString:(wphoto == nil ? @"Photo is nil" : @"Photo is not nil")];
        [ms appendString:@"\n"];
        if(wphoto != nil) {
                [ms appendString:wphoto.title];
        }
        self.resultLabel.text = ms;
    DDLogDebug(@"%s exit", __PRETTY_FUNCTION__);
}
```

2.7.4　输出分析

图 2-7 显示了输出。



```
2014-08-17 23:32:48.391 HPerf Apps[12359:60b] Flurry: Starting session on Agent Version [Flurry_iOS_138_5.2.0]
2014-08-17 23:32:48.719 HPerf Apps[12359:60b] FV_Appear

2014-08-17 23:32:54.754 HPerf Apps[12359:60b] [D] [enter] createStrongPhoto
2014-08-17 23:32:54.756 HPerf Apps[12359:60b] [D] Strong Photo: <HPPhoto: 0x16dd0e30>
2014-08-17 23:32:54.757 HPerf Apps[12359:60b] [D] [exit] createStrongPhoto
2014-08-17 23:32:54.757 HPerf Apps[12359:60b] [V] HPVPhoto dealloc-ed

2014-08-17 23:32:56.821 HPerf Apps[12359:60b] [D] [enter] createWeakPhoto
2014-08-17 23:32:56.822 HPerf Apps[12359:60b] [V] HPVPhoto dealloc-ed
2014-08-17 23:32:56.823 HPerf Apps[12359:60b] [D] Weak Photo: (null)
2014-08-17 23:32:56.824 HPerf Apps[12359:60b] [D] [exit] createWeakPhoto

2014-08-17 23:32:58.570 HPerf Apps[12359:60b] [D] [enter] createStrongToWeakPhoto
2014-08-17 23:32:58.572 HPerf Apps[12359:60b] [D] Strong Photo: <HPPhoto: 0x16dc4020>
2014-08-17 23:32:58.573 HPerf Apps[12359:60b] [D] Weak Photo: <HPPhoto: 0x16dc4020>
2014-08-17 23:32:58.574 HPerf Apps[12359:60b] [D] [exit] createStrongToWeakPhoto
2014-08-17 23:32:58.575 HPerf Apps[12359:60b] [V] HPVPhoto dealloc-ed

2014-08-17 23:33:00.487 HPerf Apps[12359:60b] [D] [enter] createUnsafeUnretainedPhoto
2014-08-17 23:33:00.489 HPerf Apps[12359:60b] [V] HPVPhoto dealloc-ed
2014-08-17 23:33:00.490 HPerf Apps[12359:60b] [D] Unsafe Unretained Photo: <HPPhoto: 0x16ea26a0>
2014-08-17 23:33:00.490 HPerf Apps[12359:60b] [D] [exit] createUnsafeUnretainedPhoto

2014-08-17 23:33:14.304 HPerf Apps[12359:60b] [D] [enter] createUnsafeUnretainedPhoto
2014-08-17 23:33:14.305 HPerf Apps[12359:60b] [V] HPVPhoto dealloc-ed
2014-08-17 23:33:14.306 HPerf Apps[12359:60b] [D] Unsafe Unretained Photo: Unsafe Unretained Photo:
(lldb)
```


图 2-7：变量的生命周期限定符

结果不言自明，我们还能观察到一些有趣的东西。

(1) __strong 引用（createStrongPhoto: 方法）确保了对象在其作用域内不会被销毁。对象只会在方法完成之后被回收。
(2) __weak 引用（createWeakPhoto: 方法）对引用计数没有贡献。因为内存被分配在方法内且一个 __weak 引用指向这段内存，所以引用计数为 0，对象被立即回收，甚至在其被用于紧邻的下一个语句前。
(3) 在 createStrongToWeakPhoto: 方法中，虽然 __weak 引用不会增加引用计数，但之前创建的 __strong 引用确保了对象不会在方法结束前释放。
(4) createUnsafeUnretainedPhoto: 方法的结果更加有趣。注意，对象会立即被释放，但由于内存还没有被回收，这个引用可以使用，且不会导致错误。
(5) 但是，当再次调用该方法时，我们不仅看到对象已经析构，而且内存也被重新分配和再使用了。于是，使用该引用导致了非法访问，应用出现了以 SIGABRT 为信号的崩溃。这是由内存在后续（对象析构之后，访问内存之前）被回收使用造成的。

观察图 2-8，你会发现内存刚好在设置 title 属性前被回收了，从而导致了 unrecognized selector sent to instance 错误。这是因为内存已经被回收，并且现在可能已经用于存储其他对象。

```
<UIKit/UIView.h> may also be helpful.
2014-08-24 15:51:40.709 HPerf Apps[85481:60b] FV_Appear
2014-08-24 15:51:43.512 HPerf Apps[85481:60b] FV_Ph_UU
2014-08-24 15:51:43.513 HPerf Apps[85481:60b] [D] [enter] createUnsafeUnretainedPhoto
2014-08-24 15:51:43.513 HPerf Apps[85481:60b] [V] HPVPhoto dealloc-ed
2014-08-24 15:51:43.513 HPerf Apps[85481:60b] [D] Unsafe Unretained Photo: <HPPhoto:
0x10ea32180>
2014-08-24 15:51:43.513 HPerf Apps[85481:60b] -[__NSCFString setTitle:]: unrecognized
selector sent to instance 0x10ea32180
```

图 2-8：__unsafe_unretained 导致的崩溃

2.8　僵尸对象

僵尸对象是用于捕捉内存错误的调试功能。

通常情况下，当引用计数降为 0 时对象会立即被释放，但这使得调试变得困难。如果开启了僵尸对象，那么对象就不会立即释放内存，而是被标记为僵尸。任何试图对其进行访问的行为都会被日志记录，因而你可以在对象的生命周期中跟踪对象在代码中被使用的位置。

NSZombieEnabled 是一个环境变量，可以控制 Core Foundation 的运行时是否将使用僵尸对象。不应长期保留 NSZombieEnabled，因为默认情况下不会有对象被真正析构，这会导致应用使用大量的内存。特别说明一点，在发布的构建包中一定要禁用 NSZombieEnabled。

要想设置 NSZombieEnabled 环境变量，需要进入 Product → Scheme → Edit Scheme。选择左侧的 Run，然后在右侧选取 Diagnostics 标签页。选中 Enable Zombie Objects 选项，如图 2-9 所示。

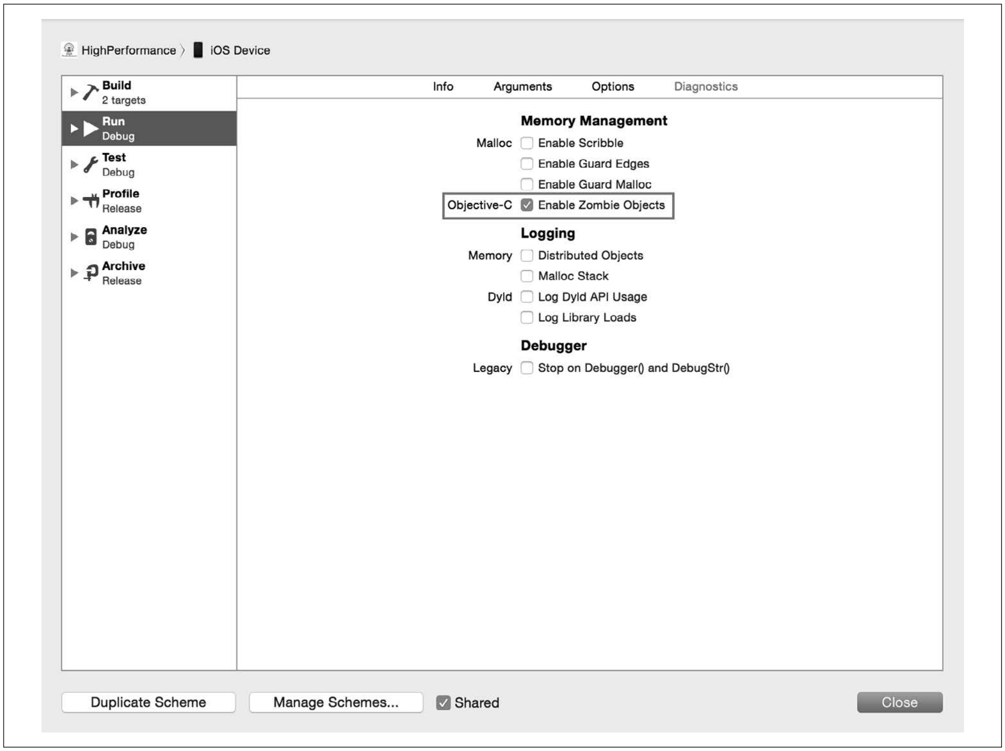


图 2-9：在 XCode 设置中开启僵尸对象

2.9　内存管理规则

现在我们已经了解了生命周期限定符的细节，复习一下内存管理的基本规则非常重要。

正如苹果公司的官方文档所述，内存管理有四个基本规则。

- 你拥有所有自己创建的对象，如 `new`、`alloc`、`copy` 或 `mutableCopy`。
- 你可以用 MRC 中的 `retain` 或者 ARC 中的 `__strong` 引用来拥有任何对象的持有关系。
- 在 MRC 中，当不再需要某个对象时，你必须立即使用 `release` 方法来放弃对该对象的持有关系。而在 ARC 中则无需任何特殊操作。持有关系会在对象失去最后的引用（如方法中的最后一行代码）时被抛弃。
- 一定不能抛弃原本并不存在持有关系的对象。

要想避免内存泄漏和应用崩溃，你应当在编写 Objective-C 代码时牢记这些规则。

2.10　循环引用

引用计数的最大陷阱在于，它不能处理环状的引用关系，即 Objective-C 的循环引用。我们将在本节中讨论循环引用出现的典型场景，并介绍避免循环引用的最佳实践。

如果仔细学习了上一节描述的规则，你会发现它们不过是引用计数的实现。声明持有关系会增加引用计数，而放弃持有关系则会减少引用计数。当引用计数降为 0 时，系统将回收对象并释放内存。

在我们的示例应用中，HPAlbum 实体包含 coverPhoto 和 photos 数组，以表示相册的封面照片以及与之关联的照片。类似地，除了其他的属性（如 URL、标题、评论等），HPPhoto 可能还代表一张照片属于某个相册。例 2-21 展示了定义这一实体信息的代表性代码。

例 2-21　循环引用

```
@class HPPhoto;

@interface HPAlbum : NSObject

@property (nonatomic, copy) NSString *name;
@property (nonatomic, strong) NSDate *creationTime;
@property (nonatomic, copy) HPPhoto *coverPhoto; ❶
@property (nonatomic, copy) NSArray *photos; ❷

@end


@interface HPPhoto : NSObject

@property (nonatomic, strong) HPAlbum *album; ❸
@property (nonatomic, strong) NSURL *url;
@property (nonatomic, copy) NSString *title;
@property (nonatomic, copy) NSArray *comments;

@end
```

❶ HPAlbum 对 coverPhoto 有一个强引用，类型为 HPPhoto。
❷ 它通过 photos 数组还持有了许多其他的 HPPhoto 对象。
❸ HPPhoto 通过强引用指向了它所属的相册。

为了简化讨论，我们假设一个相册中包含两张照片：p1（相册封面）和 p2。引用计数如下。

- p1 在 photos 和 coverPhoto 中有强引用。引用计数为 2。
- p2 在 photos 中有强引用。引用计数为 1。
- album 在 p1 和 p2 中有强引用。引用计数为 2。

我们在前面的 2.6 节中讨论过强引用。

这些对象通过名为 createAlbum 的方法被创建。虽然这些对象从某个时间点后不再被使用，但它们的内存不会被释放，因为它们的引用计数都不会降为 0。图 2-10 演示了这种关系。

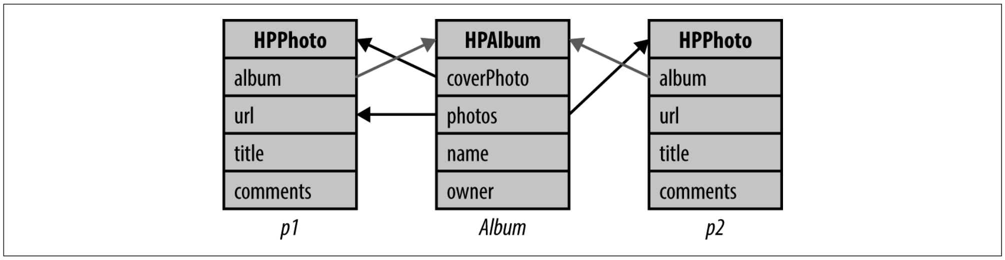


图 2-10：相册与照片的引用

2.10.1　避免循环引用的规则

上一节演示了发生循环引用的场景。在本节中，我们将关注避免在代码中发生循环引用的规则。

- 对象不应该持有它的父对象，应该用 `weak` 引用指向它的父对象（见 2.6 节）。
  在上一个场景中，照片被 `album` 所包含，我们可以将照片看成孩子。因此，从照片到相册的引用应该是弱引用。弱引用对引用计数没有贡献。

  更新后的引用计数如下。

  (1) `p1` 通过 `photos` 和 `coverPhoto` 被强引用。引用计数为 2。
  (2) `p2` 通过 `photos` 被强引用。引用计数为 1。
  (3) `album` 不存在任何强引用。引用计数为 0。

  因此，当不再被使用，`album` 对象会被回收。一旦 `album` 被释放，`p1` 和 `p2` 的引用计数会下降为 0，于是它们也会被回收。

- 作为必然的结果，一个层级体系中的子对象应该保留祖先对象。
- 连接对象不应持有它们的目标对象。目标对象的角色是持有者。连接对象包括以下几种。

  (1) 使用委托的对象。委托应该被当作目标对象，即持有者。
  (2) 包含目标和 action 的对象，这是由上一条规则推理得到的。例如，`UIButton` 会调用它的目标对象上的 action 方法。按钮不应该保留它的目标。
  (3) 观察者模式中被观察的对象。观察者就是持有者，并会观察发生在被观察对象上的变化。

- 使用专用的销毁方法中断循环引用。

  双向链表中存在循环引用，环形链表中也存在循环引用。

  在这类情况下，一旦明确对象不会再被使用时（当链表的表头超出作用范围），你要编写代码以打破链表的链接。创建一个（名为 `delink` 的）方法切断其自身与链表中下一个节点的链接。通过访问者模式递归地执行这一过程，从而避免无限递归。

2.10.2 循环引用的常见场景

大把的常见场景会导致循环引用。例如，使用线程、计时器、简单的块方法或委托都可能会导致循环引用。接下来我们将逐步探索这些场景，并给出避免循环引用的步骤。

1. 委托

委托很可能是引入循环引用的最常见的地方。在应用启动时，从服务器获取最新的数据并更新 UI 是常见的事情。当用户点击刷新按钮时也会触发类似的刷新逻辑。

考虑这个特定的场景：展示了一组记录的视图控制器，以及一个用户点击后就会刷新列表的刷新按钮。

我们在实现方面用了两个类：`HPDataListViewController` 用于 UI，`HPDataUpdateOp` 用于模拟对网络的调用。例 2-22 展示了视图控制器的代码，例 2-23 展示了更新操作的代码。

例 2-22　应用更新调用

```
//HPDataListViewController.h
@interface HPDataListViewController : UIViewController ❶

@property (nonatomic, strong) HPDataUpdateOp *updateOp; ❷
@property (nonatomic, strong) BOOL refreshing;

- (IBAction)onRefreshClick:(id)sender;

@end

//HPDataListViewController.m
@implementation HPDataListViewController

//为达到简洁的目的,删除viewDidLoad的代码

- (IBAction)onRefreshClicked:(id)sender { ❸
    DDLogDebug(@"%s enter", __PRETTY_FUNCTION__); ❹
    if([self.refreshing == NO]) {
        self.refreshing = YES;
        if(self.updateOp == nil) {
            [self.updateOp = [[HPDataUpdateOp new];
        }
        [self.updateOp startWithDelegate:self
            withSelector:@selector(onDataAvailable:)]; ❺
    }
    DDLogDebug(@"%s exit", __PRETTY_FUNCTION__); ❹
}

- (void)onDataAvailable:(NSArray *)records { ❻
    //用最新记录更新UI
    self.refreshing = NO;
    self.updateOp = nil;
}

@end
```

❶ HPDataListViewController 在列表中显示数据。
❷ updateOp 实现拉取数据的网络操作。
❸ 用户点击 refreshButton 时会调用这个方法。
❹ 记录日志以监控执行顺序。
❺ updateOp 方法可以在结果可用时调用回调方法。
❻ onDataAvailable 是回调方法。它会更新视图控制器的状态和 UI。

例 2-23　更新操作

```
//HPDataUpdateOp.m
@implementation HPDataUpdateOp

-(void)startWithDelegate:(id)delegate withSelector:(SEL)selector {
 dispatch_async(
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{ ❶
         //执行某个操作 ❷
                dispatch_async(dispatch_get_main_queue(), ^{ ❸
                 if([delegate respondsToSelector:selector]) {
                         [delegate performSelector:selector
     withObject:[NSArray arrayWithObjects:nil]; ❹
                 }
         });
        });
}

-(void)dealloc {
        DDLogDebug(@"%s called", __PRETTY_FUNCTION__);
}

@end
```

❶ 所有需要长时间运行的任务都应该在主线程之外进行。
❷ 假设操作需要消耗 2 秒。
❸ 一旦结果可用，将上下文切换回主线程，然后……
❹ ……调用选择器。

onRefreshClicked 方法将 self 传入 updateOp 方法。同时，HPDataListViewController 持有了对 updateOp 的引用。这就是产生循环引用的地方。

就解决方案而言，其中一个选择是不要将 updateOp 作为一个属性，而在 onRefreshClicked: 方法中创建 HPDataUpdateOp 的一个实例。此时，updateOp 会持有 HPDataListViewController 对象的引用，但反过来不会。更新后的代码如例 2-24 所示。

例 2-24　在无需属性的情况下更新应用

```
- (IBAction)onRefreshClicked:(id)sender {
    DDLogDebug(@"%s enter", __PRETTY_FUNCTION__);
    if(self.refreshing == NO) {
        self.refreshing = YES;
        HPDataUpdateOp *updateOp = [[HPDataUpdateOp new]; ❶
        [updateOp startWithDelegate:self withSelector:@selector(onDataAvailable:)];
    }
```

```
        DDLogDebug(@"%s exit", __PRETTY_FUNCTION__);
    }
```

❶ 创建本地变量而不再持有引用。

这确实解决了引入循环引用的问题，但却带来了其他问题。updateOp 对象永远不会被其他地方所引用，因此，一旦 onRefreshClicked: 方法执行完毕，它的引用计数将降为 0，然后立即被回收。输出如图 2-11 所示。

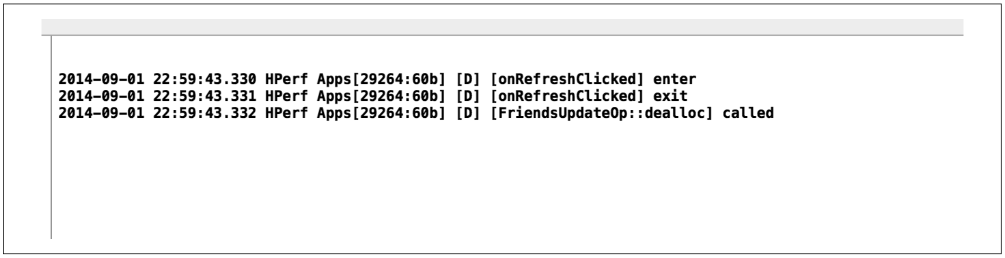


图 2-11：使用本地变量的输出结果

正如演示所示，HPDataUpdateOp 是高度简化的场景。通常来说，应用会通过网络队列来实现排队执行更新操作。并且在操作完成之后，用户很有可能会切换到其他的视图控制器。在这种场景中，视图控制器在理想的情况下应立即被回收。但因为视图控制器正被网络操作所使用，所以它不会被回收。现在想象一下多个视图控制器被队列中的操作所持有。虽然没有造成循环引用，但确实会增加内存峰值需求。因此，这毫无疑问是个 bug，因为如果没有视图控制器的存在，操作在理想状态下应该及时地释放对象。

因此，严格来讲，这个修复并没有解决问题。原因是什么呢？问题在于 HPDataUpdateOp 持有了 HPDataListViewController 对象的强引用。但这里也不能使用弱引用，因为这会导致这些对象之间完全失去链接。

解决方案是在委托（当前示例为视图控制器）中建立对操作的强引用，并在操作中建立对委托的弱引用。

当准备调用回调方法时，操作应当获取委托的的强引用。

不仅如此，我们还应该在 HPDataUpdateOp 中引入 cancel 方法，以便在视图控制器即将被回收时可以调用 cancel 方法。例 2-25 展示了实现这一效果的更新代码。

例 2-25　HPDataListViewController 和 HPDataUpdateOp 的最终版本

```
//HPDataListViewController
-(IBAction)onRefreshClicked:(id)sender {
 DDLogDebug(@"%s enter", __PRETTY_FUNCTION__);
        self.updateOp = [[HPDataUpdateOp new]; ❶
        [self.updateOp startUsingDelegate:self
            withSelector:@selector(onDataAvailable:)];
        DDLogDebug(@"%s exit", __PRETTY_FUNCTION__);
}

-(void)onDataAvailable:(NSArray *)records {
```

```
 DDLogDebug(@"%s called", __PRETTY_FUNCTION__);
        self.resultLabel.text = @"[- onDataAvailable] called";
        self.updateOp = nil; ❷
}

-(void)dealloc {
 DDLogDebug(@"%s called", __PRETTY_FUNCTION__);
        if(self.updateOp != nil) {
                [self.updateOp cancel]; ❸
        }
}

//HPDataUpdateOp.h
@protocol HPDataUpdateOpDelegate <NSObject>

-(void)onDataAvailable:(NSArray *)records;

@end

@interface HPDataUpdateOp

@property (nonatomic, weak) id<HPDataUpdateOpDelegate> delegate; ❹

-(void)startUpdate;
-(void)cancel;

@end

//HPDataUpdateOp.m
@implementation HPDataUpdateOp
-(void)startUpdate {
        dispatch_async(
  dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
                //执行网络调用,然后报告结果
  //NSArray *records = ...
                dispatch_async(dispatch_get_main_queue(), ^{
   id<HPDataUpdateOpDelegate> delegate = self.delegate; ❺
   if(!delegate) { ❻
                                return;
                        } else { ❼
                                [delegate onDataAvailable:records];
                        }
                });
        });
}

-(void)cancel { ❽
 //取消执行中的网络请求
        self.delegate = nil;
}
```

❶ 使用属性表示操作。视图控制器持有操作。

❷ 当任务完成后将属性设置为 `nil`。实现对操作对象的回收。

❸ 若视图控制器即将被回收，则取消操作。

❹ 操作保持对回调委托的弱引用。
❺ 尝试获取委托的强引用。
❻ 如果原始对象仍然存在……
❼ ……通过它报告 `onDataAvailable`。
❽ 取消操作显式地要求废弃回调对象。

本质上，这里实现了 2.10.1 节中的第一条规则。`HPDataListViewController` 是持有者，而 `HPDataUpdateOp` 是被持有的对象（如持有层级中的子节点）。

图 2-12 展示了网络响应早于用户离开页面的结果。图 2-13 展示了网络响应晚于用户离开页面时的输出信息。

```
2014-09-01 23:02:25.396 HPerf Apps[29283:60b] [D] [onRefreshClicked] enter
2014-09-01 23:02:25.398 HPerf Apps[29283:60b] [D] [onRefreshClicked] exit
2014-09-01 23:02:27.400 HPerf Apps[29283:60b] [D] [onFriendsAvailable] called
2014-09-01 23:02:27.401 HPerf Apps[29283:60b] [D] [FriendsUpdateOp::dealloc] called
```

图 2-12：操作完成后，视图控制器可用时使用更新代码的结果

```
2014-09-01 23:47:35.490 HPerf Apps[29283:60b] [D] [onRefreshClicked] enter
2014-09-01 23:47:35.491 HPerf Apps[29283:60b] [D] [onRefreshClicked] exit
2014-09-01 23:47:36.586 HPerf Apps[29283:60b] Appear_Chxx
2014-09-01 23:47:36.587 HPerf Apps[29283:60b] [D] [HPFriendsListViewController::dealloc] called
2014-09-01 23:47:37.493 HPerf Apps[29283:60b] [D] [FriendsUpdateOp::start] [dispatch_async::main]
delegate is nil
2014-09-01 23:47:37.494 HPerf Apps[29283:60b] [D] [FriendsUpdateOp::dealloc] called
```

图 2-13：操作完成前，视图控制器已销毁时使用更新代码的结果

虽然看起来似乎比较直观，但随着执行深入到不同的层而产生复杂的对象图时，情况会变得非常复杂。你需要确保自己不会在网络的底层、数据库以及用于 UI 上层（如创建对象的层）的存储中持有引用。

2. 块

与不正确地使用委托对象导致的问题类似，在使用块时，捕获外部变量也是导致循环引用的原因。

参考例 2-26 中的简单代码。

例 2-26　使用块捕获变量

```
-(void)someMethod {
    SomeViewController *vc = [[SomeViewController alloc] init];
    [self presentViewController:vc animated:YES
        completion:^{
```

```
            self.data = vc.data;
            [self dismissViewControllerAnimated:YES completion:nil];
        }];
}
```

遗憾的是，这又将产生历时长久的对象——子视图控制器不会被销毁，因为正在向用户显示它，其父视图控制器也不会被回收，因为它被 completion 块捕获了。在 SomeViewController 执行耗时较长的任务时，如图像处理或复杂的视图渲染，其父视图控制器的内存不会被清空，应用存在内存不足的风险。

例 2-27 所示的解决方案与我们在前一节讨论的内容相似。

例 2-27　使用块时的变量捕获

```
-(void)someMethod {
    SomeViewController *vc = [[SomeViewController alloc] init];

    __weak typeof(self) weakSelf = self; ❶

    [self presentViewController:vc animated:YES
        completion:^{
            typeof(self) theSelf = weakSelf; ❷

            if(theSelf != nil) { ❸
                theSelf.data = vc.data; ❹
                [theSelf dismissViewControllerAnimated:YES completion:nil];
            }
        }];
}
```

❶ 获得一个弱引用。
❷ 通过弱引用获得强引用。注意，__strong 是隐式的，可以增加引用计数……
❸ ……只在不为 nil 时才继续……
❹ ……处理后续操作。

3. 线程与计时器

不正确地使用 NSThread 和 NSTimer 对象也可能会导致循环引用。运行异步操作的典型步骤如下。

- 如果没有编写更高级的代码来管理自定义的队列，则在全局队列上使用 dispatch_async 方法。
- 在需要的时间和地点用 NSThread 开启异步执行。
- 使用 NSTimer 周期性地执行一段代码。

假设有一个新闻应用，其 UI 显示已登录用户的新闻流，且该应用每隔 2 分钟自动刷新一次。

例 2-28 展示了执行周期性更新的典型代码。

例 2-28　使用 NSTimer

```
@implementation HPNewsFeedViewController
```

```
-(void)startCountdown {
 self.timer = [NSTimer scheduledTimerWithTimeInterval:120
  target:self
  selector:@selector(updateFeed:)
  userInfo:nil repeats:YES];
}

-(void)dealloc {
    [self.timer invalidate];
}

@end
```

例 2-28 中的循环引用非常明显——对象持有了计时器，同时计时器也持有了对象。与例 2-22 类似，我们无法通过“消灭”属性来解决问题。事实上，我们需要持有 `timer` 属性，以便其可以在后续被销毁。

对这段代码来说，运行循环也将持有计时器，并直到 `invalidate` 方法被调用时才会释放它。

这就创建了对计时器对象的附加持有引用，即使代码中并没有显式的引用关系，这仍然会导致循环引用。

> `NSTimer` 对象导致了被运行时持有的间接引用。这些引用是强引用，因而目标的引用计数会以 2（而不是 1）增长。必须对计时器对象调用 `invalidate`，以移除引用。

假设例 2-28 中的代码属于一个*视图控制器*，并且由于用户的操作，视图控制器在应用中被创建了多次。可以想象内存泄漏的总量会有多么大。

如果你使用的是 `NSThread`，那也不要得意，因为同样的问题还是会发生。这个问题有两个解决方法。

- 主动调用 `invalidate`。
- 将代码分离到多个类中。

接下来我们将分别讨论这两种解决方法。

别指望 `dealloc` 能够清理这些对象。为什么呢？如果建立了循环引用，那 `dealloc` 方法永远都不会被调用，计时器也永远都不会执行 `invalidated`。因为运行循环会跟踪*活跃*的计时器对象和线程对象，所以仅在代码中置为 `nil` 并不能销毁对象。要想解决这个问题，可以创建一个自定义方法，以更加明确的方式执行清理操作。

在一个视图控制器中，调用这个清理方法的最佳时机是用户离开视图控制器的时候，这个时机既可以是点击返回按钮，也可以是其他类似的行为（类知道此事发生的地方）。我们将这个方法命名为 `cleanup`。例 2-29 提供了实现的代码。

例 2-29　清理 `NSTimer`

```
-(void)didMoveToParentViewController:(UIViewController *) parent { ❶
```

```
  if(parent == nil) {
   [self cleanup];
  }
 }

-(void)cleanup {
 [self.timer invalidate];
}
```

❶ 当视图控制器进入或离开父视图控制器时，调用 `didMoveToParentViewController` 方法。

在例 2-29 中，通过重写 `didMoveToParentViewController` 方法，当用户从父视图控制器离开当前视图控制器时，我们执行清理操作。这要比调用 `dealloc` 更加明确。

另一种方法是修改返回按钮的目标，如例 2-30 所示。

例 2-30　通过拦截返回按钮执行清理

```
-(id)init {
    if(self = [super init]) {
        self.navigationItem.backBarButtonItem.target = self;
        self.navigationItem.backBarButtonItem.action
            = @selector(backButtonPressDetected:); ❶
    }
    return self;
}

-(void)backButtonPressDetected:(id)sender {
    [self cleanup]; ❷
    [self.navigationController popViewControllerAnimated:TRUE];
}
```

❶ 拦截导航控制器对返回按钮的点击事件。
❷ 在视图控制器弹出之前进行清理。

另一个清理方案是将持有关系分散到多个类中——任务类执行具体动作，所有者类调用任务。

我更推荐后一个方案，理由如下。

- 清理器有定义良好的职责持有者。
- 需要时任务可以被多个持有者重复使用。

我们可以将前面的代码拆成两个类：`HPNewsFeedViewController` 展示最新的 feed 流，而 `HPNewsFeedUpdateTask` 周期性地执行，检查填充视图控制器的最新的 feed 流。

要实现这个效果，重构后的代码如例 2-31 所示。

例 2-31　使用计时器重构后的代码

```
//HPNewsFeedUpdateTask.h
@interface HPNewsFeedUpdateTask

@property (nonatomic, weak) id target; ❶
@property (nonatomic, assign) SEL selector;
```

```
@end

//HPNewsFeedUpdateTask.m
@implementation HPNewsFeedUpdateTask

-(void)initWithTimeInterval:(NSTimeInterval)interval
    target:(id)target selector:(SEL)selector { ❷

    if(self = [super init]) {
        self.target = target;
        self.selector = selector;

        self.timer = [NSTimer scheduledTimerWithTimeInterval:interval
            target:self selector:@selector(fetchAndUpdate:)
            userInfo:nil repeats:YES];
    }
    return self;
}

-(void)fetchAndUpdate:(NSTimer *)timer { ❸
    //检索feed
    HPNewsFeed *feed = [self getFromServerAndCreateModel];
    __weak typeof(self) weakSelf = self; ❹

    dispatch_async(dispatch_get_main_queue(), ^{
        __strong typeof(self) sself = weakSelf;
        if(!sself) {
            return;
        }

        if(sself.target == nil) {
            return;
        }

        id target = sself.target; ❺
        SEL selector = sself.selector;

        if([target respondsToSelector:selector]) {
            [target performSelector:selector withObject:feed];
        }
    });
}

-(void)shutdown { ❻
    [self.timer invalidate];
    self.timer = nil;
}
@end

//HPNewsFeedViewController.m
@implement HPNewsFeedViewController

-(void)viewDidLoad { ❼
    self.updateTask = [HPNewsFeedUpdateTask initWithTimeInterval:120
```

```
        target:self selector:@selector(updateUsingFeed:)];
}

-(void)updateUsingFeed:(HPNewsFeed *)feed { ❽
    //更新UI
}

-(void)dealloc { ❾
    [self.updateTask shutdown];
}
@end
```

接下来我们看一下对 HPNewsFeedUpdateTask 的详细分析。

(1) target 属性 ❶ 是弱引用。target 会在这里实例化任务并持有它。
(2) initWithTimeInterval:❷ 是推荐使用的方法。它需要一些必要的输入，并启动计时器。
(3) fetchAndUpdate: 方法 ❸ 会周期性地执行。
(4) 在使用异步块时需要确保不会引入循环引用。我们在块方法内使用 __weak 引用 ❹。
(5) 在 fetchAndUpdate: 方法 ❼ 中，target 和 selector 的本地变量都在调用 respondsToSelector: 前创建，并执行操作。
    这样做是为了避免在以下的执行序列中发生竞争情况。
    a. 在某个线程 A 中调用 [target respondsToSelector:selector]。
    b. 在线程 B 中修改 target 或 selector。
    c. 在线程 A 中调用 [target performSelector:selector withObject:feed]。有了这个代码，即使 target 或 selector 此刻已经发生改变，performSelector 仍然会被正确的 target 和 selector 所调用。
(6) shutdown 方法 ❺ 对计时器调用 invalidate。运行循环会终止对计时器的调用，于是计时器成为任务对象持有的唯一引用。

从使用方面来看，HPNewsFeedViewController 使用了 HPNewsFeedUpdateTask。控制器没有被除父控制器之外的对象所持有。因此，当用户离开页面时（也就是点击了返回按钮时），引用计数会降为 0，视图控制器会被销毁。这反过来会导致更新任务停止，进而导致计时器被设定无效，从而触发所有关联对象（包括 timer 和 updateTask）的析构链。

现在我们对例 2-31 中的 HPNewsFeedViewController 代码进行分析。

(1) 在 viewDidLoad 方法 ❼ 中，对任务对象进行初始化，其内部会触发计时器。
(2) updateUsingFeed: 是 HPNewsFeedUpdateTask 对象周期性调用的回调方法。
(3) dealloc 方法 ❽ 负责调用任务对象的 shutdown 方法，其内部会销毁计时器。注意，dealloc 在此处是明确可用的，因为该对象没有被其他的地方所引用。

当使用 NSTimer 和 NSThread 时，总是应该通过间接的层实现明确的销毁过程。这个间接层应使用弱引用，从而保证所拥有的对象能够在停止使用后执行销毁动作。

2.10.3 观察者

与使用委托和订阅复杂数据变化的回调不同，系统提供了两种内置的可用选择，以便监听变化。之所以说它们是*内置的*，主要是因为我们无需编写任何自定义的代码来跟踪观察者——运行时对管理观察者提供了支持。这类技术包括：

- 键 – 值观察
- 通知中心

1. 键–值观察

Objective-C 允许用 `addObserver:forKeyPath:options:context:` 方法在任何 `NSObject` 子类的对象上添加观察者。观察者会通过 `observeValueForKeyPath:ofObject:change:context:` 方法得到通知。`removeObserver:forKeyPath:context:` 方法用于解除注册或移除观察者。这就是众所周知的*键 – 值观察*。

这是一个极为有用的特性，尤其是在以调试为目的跟踪某些共享于应用多个部分（如用户接口、业务逻辑、持久化以及网络）的对象时。

举一个相关的示例，类似的对象可能是某个自定义类，它持有着应用当前的状态细节。例如，标识用户是否登录、已登录的用户信息、电子商务应用中购物车的内容、或者用户在信息应用内最新消息的接收人。为了方便调试，你可能会为这个对象添加一个观察者，以跟踪所有的修改或更新。

键 – 值观察在双向数据绑定中也非常有用。视图可以关联委托来响应那些会导致模型更新的用户交互。键 – 值观察可以用于反向的绑定，以便在模型发生变化时更新 UI。

以下内容摘自官方文档（https://sites.google.com/site/appleiotemp//1IBd0lC）。

> 键 – 值观察方法 `addObserver:forKeyPath:options:context:` 不会维持观察对象、被观察对象及上下文对象的强引用。如有必要，你需要自行维护对它们的强引用。

这意味着观察者需要有足够长的生命周期才能够持续地监控变化。你需要额外关注观察者的生命周期，而且要持续到所观察的内存被废弃之后。

例 2-32 用集中式的 `ObserverManager` 类来实现键 – 值观察，`ObserverManager` 类会返回一个与持有者相关的 `ObserverObserveeHandle` 实例。当观察启动程序（示例中为视图控制器）需要观察 `keyPath`，它会调用 `addObserverToObject:forKey:` 方法并储存 `ObserverObserveeHandle`，从而实现与视图控制器同时销毁。`handle` 会在其销毁期间移除观察者。

这里尝试解决的问题本质上与 `NSTimer` 的例子相似。只是计时器的场景需要建立一个弱引用，而这里如果没有正确处理，则会导致观察者永久被销毁。

例 2-32　键 – 值观察

```
@interface ObserverObserveeHandle

@property (nonatomic, strong) MyObserver *observer;
@property (nonatomic, strong) NSObject *obj;
@property (nonatomic, copy) NSString *keyPath;
```

```
-(id)initWithObserver:(MyObserver *)observer
 target:(NSObject *)obj
 keyPath:(NSString *)keyPath;

@end

@implementation ObserverObserveeHandle

-(id)initWithObserver:(MyObserver *)observer
 target:(NSObject *)obj
 keyPath:(NSString *)keyPath {
  //删除以便更加简洁
}

-(void)removeObserver {
 [self.obj removeObserver:self forKeyPath:self.keyPath context:nil];
 self.obj = nil;
}

-(void)dealloc {
 [self removeObserver];
}
@end

@interface ObserverManager
//删除以便更加简洁
@end

@implementation ObserverManager
NSMutableArray *observers;

+(ObserverObserveeHandle)addObserverToObject:(NSObject *)obj
 forKey:(NSString *)keyPath {
 MyObserver *observer = [[MyObserver alloc] init];
 [obj addObserver:observer forKeyPath:keyPath
  options:(NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld)
  context:NULL];

 ObserverObserveeHandle *details = [[ObserverObserveeHandle alloc]
  initWithObserver:observer target:obj keyPath:keyPath];
 [observers addObject:details];

 return details;
}

@interface SomeViewController

@property (nonatomic, strong) IBOutlet UILabel *resultLabel;
@property (nonatomic, strong) ObserverObserveeHandle *resultLabelMonitor;

@end

@implementation SomeViewController
```

```
-(void)viewDidLoad {
 self.resultLabelMonitor = [ObserverManager
   addObserverToObject:self.nameTextField
   forKey:@"text"];
}

@end
```

> 当你为目标对象添加键 – 值观察者时，目标对象的生命周期至少应该和观察者一样长，因为只有这样才有可能从目标对象移除观察者。这可能会导致目标对象的生命周期比预期要长，也是你需要额外小心的地方。

例 2-32 似乎提供了一个优秀的解决方案，通过风格良好且不会出错的代码，该示例解除了执行清理带来的负担。然而，这里仍然存在问题，就是观察者的所有通知都会执行相同的代码片段（如 `Observer` 类定义的代码）。

如何解决这个问题呢？这很值得思考。提示：使用*块方法*。在块方法内，如果需要调用使用 `self` 的方法，别忘了在将它用于块方法内部的使用前先创建对 `self` 的弱引用。

于是，注册观察者的代码得到了更新，如例 2-33 所示。

例 2-33　使用块方法的键 – 值观察者

```
@implementation SomeViewController

-(void)viewDidLoad
{
    __weak typeof(self) weakSelf = self;
    self.resultLabelMonitor = [ObserverManager
        addObserverToObject:self.nameTextField
        forKey:@"text" block:^(NSDictionary *changes) {

            typeof(self) sSelf = weakSelf;
            if(sself) {
                NSLog(@"Text changed to %@",
                    [changes objectForKey:NSKeyValueChangeNewKey]);

                //需要时使用sSelf
                sSelf.resultLabel.text = @"Name changed";
            }
        }];
}

@end
```

2. 通知中心

另一个方案是使用通知中心。一个对象可以注册为通知中心（`NSNotificationCenter` 对象）的观察者，并接收 `NSNotification` 对象。与键 – 值观察者相似，通知中心不会对观察者持有强引用。这意味着开发人员得到了解放，无需为观察者的析构过早或过晚而操心。

具体的解决模式与我们在前一节讨论的内容相似。

2.10.4 返回错误

当用某个方法接收 NSError ** 参数，并在发生错误时填充错误变量，则必须使用 __autoreleasing 限定符。使用这一范式的最常见场景是需要处理输入并返回可能包含错误的值的时候。

例 2-34 展示了典型情况的方法签名。

例 2-34　返回错误

```
-(Matrix *)transposeMatrix:(Matrix *)matrix error:(NSError * __autoreleasing *)
error
{
    //处理
    //如果发生错误
    *error = [[NSError alloc] initWithDomain:@"transpose" code:123 userInfo:nil];
}
```

需要关注这一语法。关键字 __autoreleasing 要塞到两个星号之间。要牢记这一点：

```
NSError* __autoreleasing *error;
```

你可能已经注意到了，变量和属性的限定符有重要的作用，可以帮助管理和精确控制对象的生命周期，确保它们既不会太长也不会过短。每当存有疑问时，你都应该翻回去看看画板、回顾一下基础知识，并合理地定义属性和变量。有时你需要用强引用修饰创建的属性并延长其生命周期，而有时你却需要使用弱引用并确保获取适当的内存使用，以避免内存泄漏。

2.11 弱类型：id

许多场景都需要使用 id 类型。在 Cocoa framework 中见到此类型也绝非罕见。例如，在 Xcode 生成的代码中，IBAction 方法使用了 id 类型的参数来表示 sender。

另一个场景是使用 NSArray 中的对象。[6] 思考例 2-35 中的代码。

例 2-35　使用 NSArray 中的对象

```
@interface HPDataListViewController
    : UITableViewController <UITableViewDataSource, UITableViewDelegate>

@property (nonatomic, copy) NSArray *input;

@end

@implementation HPDataListViewController

-(void)tableView:(UITableView *)tableView
```

注 6：iOS 9 introduces lightweight generics for Objective-C collections for interoperability with Swift. See iOS Developer Library, “Lightweight Generics”.（https://developer.apple.com/library/content/documentation/Swift/Conceptual/BuildingCocoaApps/InteractingWithObjective-CAPIs.html#//apple_ref/doc/uid/TP40014216-CH4-ID35）

```
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSUInteger value = [self.input objectAtIndex:indexPath.row].someProperty;
    //继续
}

@end
```

在 tableView:didSelectRowAtIndexPath: 方法中，我们将某种类型（假设为 ClassX，其中包含一个属性 someProperty）的对象组成的数组作为输入。

这段代码看起来很好，而且如果尝试运行，你也很可能会得到正确的结果。我们知道不同下标的对象一直都会响应 someProperty 的选择器，所以这段代码会按照预期工作。但如果某个对象没有响应这个选择器，那么就会导致应用崩溃。

假设编译器并不需要知道类型信息，因为运行时知道应该调用哪个对象和方法。但事实是，编译器确实需要了解细节情况——具体来说，它需要知道所有参数的大小和返回结果的类型，这样才能生成正确的指令，正确地在栈上压入和弹出数据。举例来说，如果方法有两个 int 类型的参数，则需要在栈上压入 8 字节。

通常情况下，我们无需关注这些细节。在获取参数信息时，编译器会根据所调用的方法名称遍历查找导入的头文件以匹配方法名，然后通过找到的第一个匹配的方法获取参数长度。

这个方案好在适用于绝大多数情况。但如果多个类有完全相同的方法签名（如名称和参数），那么将无法正常工作。

考虑以下这个场景：在编译时，编译器没有聚焦在 ClassX 上，假设是在 ClassY 对象上。那么方法可能不会返回 NSUInteger，而可能会返回 NSInteger，甚至返回 NSString。在另外一个场景中，我们原本预期返回 NSUInteger，而实际返回了某种需要我们对其进行 invalidate 或 cleanup 的对象（如 CGColor 或 CGContext），这最终会导致内存泄漏。

问题的解决方案

为何会发生类型匹配错误呢？编译器怎么会如此不成熟？它进行了艰苦卓绝的工作来解决对象的消息发送。编译器负责生成精确的指令（例如，传递给 objc_msgSend 方法正确的值）。

好在编译器解决类型匹配错误的问题并不困难。解决方案分为两个部分。

首先，我们必须对编译器进行配置，当在 id 对象上发现多个选择器匹配时，编译器要报告错误。这是由 Strict Selector Matching 选项控制的，默认为关闭状态。与之对应的编译器参数是 -Wstrict-selector-match。当编译器发现两个选择器有不同的参数或返回类型时，将其打开可以发出警告。

图 2-14 展示了 Xcode 中的工程设置。

与此选项有关的一些问题：

- 内置的框架会产生许多警告，尽管绝大多数警告并不会带来任何麻烦；

- 如果是处理类而不是对象，那么你仍然无法发现这种问题；
- 如果导入的头文件没有正确的定义，那么这个选项也不会有任何帮助。

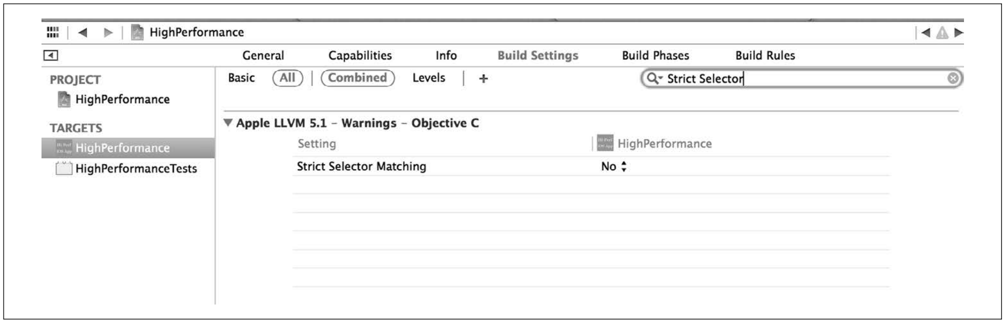


图 2-14：在 XCode 中设置严格的选择器匹配

这将我们引入了解决方案的第二部分：为编译器提供足够多的信息，以供其生成基于正确类型的信息。你可以使用强类型（例中所使用的是 ClassX）。例 2-36 展示了对代码的改动。

例 2-36　使用强类型

```
-(void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath

{
    ClassX *item =  (ClassX *) [self.input objectAtIndex:indexPath.row];
    NSUInteger value = item.someProperty;
    //继续
}
```

简而言之，在使用常规命名的方法时，应避免使用 id。尽量使用具体的类取而代之。

2.12　对象寿命与泄漏

对象在内存中活动的时间越长，内存不能被清理的可能性就越大。所以应当尽可能地避免出现长寿命的对象。当然，你需要保留代码中关键操作对象的引用，为的是不必每次都浪费时间来创建它们。尽量在使用这些对象时完成对它们的引用。

长寿命对象的常见形式是单例。日志器是典型的例子——只创建一次，从不销毁。我们会在下一节深入讨论此类情况。

另一个方案是使用全局变量。全局变量在程序开发中是可怕的东西。



要想合理地使用全局变量，必须满足以下条件：

- 没有被其他对象所持有；
- 不是常量；
- 整个应用中只有一个，而不是每个组件一个。

如果某个变量不符合这些要求，那么它不应该被用作全局变量。

复杂的对象图使得回收内存的机会变得更少，同时增加了应用因内存耗尽而崩溃的风险。如果主线程总是被迫等待子线程的操作（如网络或数据库存取），那么应用的响应性能会变得很差。

2.13 单例

单例模式是限制一个类只初始化一个对象的一种设计模式。在实践中，初始化常常在应用启动不久后执行，而且这些对象不会被销毁。

让一个对象有着与应用一样长的生命周期可不是什么好主意。如果这个对象是其他对象的源头（如一个服务定位器），若定位器的实现不正确则有可能造成内存风险。

毫无疑问，单例是必要的。但单例的实现对其使用方式有重要影响。

在充分讨论单例引入的问题之前，我们不妨先更好地理解单例，了解一下为什么确实需要使用单例。

单例极为有用，尤其是在某个系统确定只需要一个对象实例时。应该在以下情形中使用单例：

- 队列操作（如日志和埋点）
- 访问共享资源（如缓存）
- 资源池（如线程池或连接池）

一旦创建，单例会一直存活到应用关闭。日志器、埋点服务以及缓存都是使用单例的合理场景。

更重要的是，单例通常会在应用启动时进行初始化，打算使用单例的组件需要等它们准备得当。这会增加应用的启动时间。

有办法解决吗？现在还没有。如果越来越多地使用一些现成的第三方组件，内存窘迫的情况会不断加剧，尤其是在没有它们的源码的情况下。

你可以使用以下的指导原则。

- 尽可能地避免使用单例。
- 识别需要内存的部分，如用于埋点的内存缓冲区（在尚未将数据同步到服务器前使用）。寻求减少内存的方法。注意，你需要将减少内存与其他事情做权衡。减小缓冲区意味着更多的服务器通信。
- 尽量避免对象级的属性，因为它们会与对象共存亡。尽量使用本地变量。

依赖注入

完全避免单例可能并不容易，但至少可以尽量避免直接使用它们。应避免编写例 2-37 这样的代码。

例 2-37　不合理地使用依赖

```
-(void)someMethod {
    XXSomeClass *obj = [XXSomeClass sharedInstance]; ❶
    NSString *someValue = [obj operation:@"some parameter"];
    //继续
}
```

❶ someMethod 为 operation 方法使用了 XXSomeClass。

在例 2-37 中，someMethod 依赖了外部的类 XXSomeClass，不受应用设置的控制管理。这一切工作良好，但存在一些潜在的问题。

- 如果类 XXSomeClass 需要某些初始化，someMethod 假设它已经完成了初始化。然而，事实上 XXSomeClass 并不为上游使用 someMethod 的方法所知，这可能会导致 XXSomeClass 尚未初始化。
- 如果类 XXSomeClass 持有了一些资源，那么它会持续持有，哪怕 sharedInstance 后续不会再被调用。

要想避免此类陷阱，请使用依赖注入。依赖注入本质上是在需要时传递依赖。取决于依赖的范围，依赖注入可以通过自定义的初始化器或调用方法实现注入。

如果被依赖的对象在类中的多个地方被使用，那么执行注入的最佳地方就是自定义的初始化器。如果依赖的对象只在少量几个操作中，且能为这些操作提供不同的实例，那么应该对每个方法分别注入。

例 2-38 中的更新代码演示了依赖注入的两种方案：使用了 Typhoon（http://typhoon-framework.org）和 Objection（http://objection-framework.org），二者是常见且开发活跃的依赖注入框架。

例 2-38　使用依赖注入的更新代码

```
@interface MyClass

-(instancetype)initWithSomeClass:(XXSomeClass *)someClass; ❶

-(void)someMethod;
-(void)anotherMethodWithAnotherClass:(AnotherClass *)anotherClass; ❷

@end

@interface MyClass ()

@property (nonatomic, strong) XXSomeClass *someClass;
```

```
@end

@implementation MyClass

-(instancetype)initWithSomeClass:(XXSomeClass *)someClass {
    if(self = [super init]) {
        self.someClass = someClass;
    }
    return self;
}

-(void)someMethod { ❸
    NSString *someValue = [self.someClass operation:@"some parameter"];
    //继续
}

-(void)anotherMethodWithAnotherClass:(AnotherClass *)anotherClass { ❹
    NSString *someValue = [self.someClass operation:@"some parameter"];
    NSString *anotherValue = [anotherClass anotherOp:@"another parameter"];
    //继续
}

@end
```

❶ 自定义需要传入 XXSomeClass 对象的初始化器。

❷ anotherMethodWithAnotherClass 使用了 AnotherClass 对象，并需要它以参数形式传入。

❸ someMethod 现在通 someClass 属性调用 operation 方法。

❹ anotherMethodWithAnotherClass 现在可以使用 someClass 和 AnotherClass 类型的另一个对象来完成任务。

2.14 找到神秘的持有者

就算类设计精良，对象被良好持有，是否会发生内存泄漏还是不确定的。如果发生内存泄漏，获取引用图是一个不错的解决办法。那么问题来了，是否能找出一个对象上的全部引用呢？

答案位于 ARC 之前的方法 `retain`。我们需要计算这个方法的调用次数。ARC 禁止覆盖或调用这个方法，但你可以暂时在工程中禁用 ARC（详情参见图 2-15）。

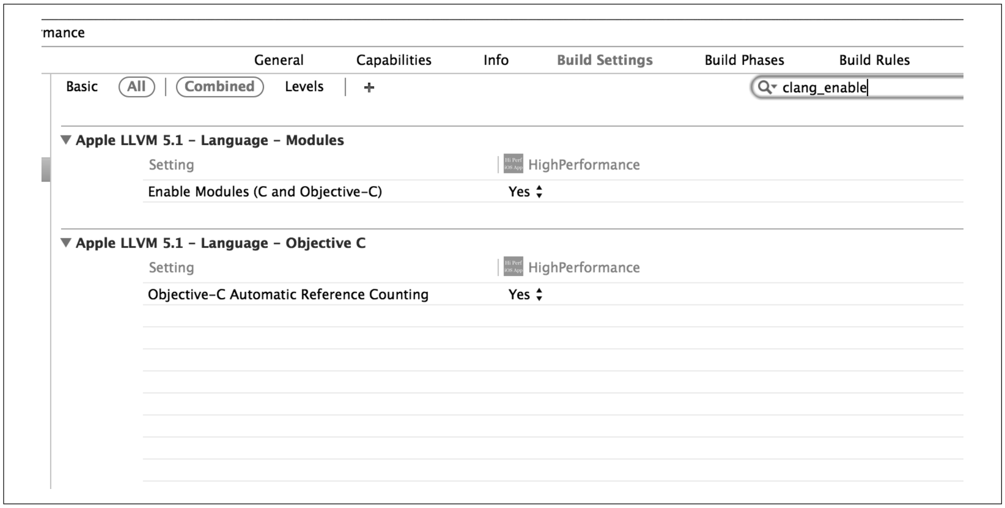


图 2-15：在工程中禁用 ARC

然后将例 2-39 中的代码添加到所有的自定义类中。这段代码不仅会记录对 `retain` 方法的调用情况，还会将调用栈打印出来。因此，除了调用次数，你还可以得到精确的调用明细。

例 2-39　使用 `retain` 获取引用计数

```
#if !__has_feature(objc_arc)
-(id) retain
{
    DDLogInfo(@"%s %@", __PRETTY_FUNCTION__, [NSThread callStackSymbols]);
        return [super retain];
}
#endif
```

2.15　最佳实践

通过遵循这些最佳实践，你将很大程度上避免许多麻烦，如内存泄漏、循环引用和较大内存消耗。（你可以将这一部分打印出来，挂在工位上，以便快速查看。）

- 避免大量的单例。具体来说，不要出现上帝对象（如职责特别多或状态信息特别多的对象）。这是一个反模式，指代一种常见解决方案的设计模式，但很快产生了不良效果。日志器、埋点服务和任务队列这样的辅助单例都是很不错的，但全局状态对象不可取。
- 对子对象使用 `__strong`。
- 对父对象使用 `__weak`。
- 对使引用图闭合的对象（如委托）使用 `__weak`。
- 对数值属性（`NSInteger`、`SEL`、`CGFloat` 等）而言，使用 `assign` 限定符。
- 对于块属性，使用 `copy` 限定符。

- 当声明使用 NSError ** 参数的方法时，需要使用 __autoreleasing，并要注意用正确的语法：NSError * __autoreleasing *。
- 避免在块内直接引用外部的变量。在块外面将它们 weakify，并在块内再将它们 strongify。参见 libextobjc 库（https://github.com/jspahrsummers/libextobjc）来了解 @weakify 和 @strongify。
- 进行必要清理时遵循以下准则：
  - 销毁计时器
  - 移除观察者（具体来说，移除对通知的注册）
  - 解除回调（具体来说，将强引用的委托设置为 nil）

2.16　生产环境的内存使用情况

注意，无论你对 Xcode 做了什么设置，它只会对调试应用起作用。当一款应用还没有真正流行起来，只有数万而不是数百万用户时，通常无需关注这些设置所导致的额外变化。

要想在不同的情境中分析应用，你可以使用埋点。定期将应用的信息发送给服务器，发送的信息包括内存的消耗，尤其是如果内存超出阈值时配合一些辅助定位信息是很不错的选择。

例如，如果内存消耗超过了 40MB，你可能需要向服务器发送一些用户所在页面的细节信息以及所执行的关键操作。另一个选择是跟踪内存的消耗，以一定的间隔在本地记录日志，然后再将数据上报给服务器。你可以用例 2-40 中的代码来找到已经使用和可用的内存。

> 在出现低内存警告时对应用进行埋点，包括内存的使用及统计信息，并在应用重新运行时将这些信息上报给服务器。使用这些信息来识别出应用发生内存溢出的常见场景和边缘情况。

例 2-40　跟踪可用和已用的内存

```
//HPMemoryAnalyzer.m

#import <mach/mach.h>

vm_size_t getUsedMemory() {
 task_basic_info_data_t info;
 mach_msg_type_number_t size = sizeof(info);
 kern_return_t kerr = task_info(mach_task_self(), TASK_BASIC_INFO,
  (task_info_t) &info, &size);

 if(kerr == KERN_SUCCESS) {
  return info.resident_size;
 } else {
  return 0;
 }
```

```
}

vm_size_t getFreeMemory() {
 mach_port_t host = mach_host_self();
 mach_msg_type_number_t size = sizeof(vm_statistics_data_t) / sizeof(integer_t);
 vm_size_t pagesize;
 vm_statistics_data_t vmstat;

 host_page_size(host, &pagesize);
 host_statistics(host, HOST_VM_INFO, (host_info_t) &vmstat, &size);

 return vmstat.free_count * pagesize;
}
```

> **使用 Instruments 进行内存分析**
>
> 你可以使用 Xcode Instruments 工具对应用的内存使用进行分析。我们将在 11.2 节深入介绍 Instruments 工具。我们对其中的 Allocations 和 Leaks 尤其感兴趣。

2.17 小结

现在你已经深入了解了 iOS 运行时是如何管理内存的，并知道了避免发生循环引用（内存泄漏的最大单一来源）的基本规则。现在你可以减少内存的消耗、降低平均和峰值内存需求了。

你可以使用僵尸对象来跟踪多次释放的对象（造成应用崩溃的最主要原因之一）。

本章还提供了用于跟踪内存使用情况的代码，你不仅可以在实验室的测试环境中运行这些代码，还可以在生产环境中运行。

第 3 章

能耗

如今的移动设备早已无处不在，待机时间成为影响设备最终销量的一个重要因素。类似地，电量消耗也是决定某个应用是否会被安装的重要因素之一。

设备中的每个硬件模块都会消耗电量。电量的最大消费者是 CPU，但这只是系统的一个方面。一个编写良好的应用需要谨慎地使用电能。用户往往会删除耗电量大的应用。

除 CPU 外，耗电量高、值得关注的硬件模块还包括：网络硬件、蓝牙、GPS、麦克风、加速计、摄像头、扬声器和屏幕。

本章重点关注消耗电量的关键领域，以及如何降低电量的消耗。我们将学习如何编写能判断电池的剩余电量及充电状态的应用，还将讨论如何在 iOS 应用中分析电源、CPU 和资源的使用。

3.1 CPU

不论用户是否正在直接使用，CPU 都是应用所使用的主要硬件。在后台操作和处理推送通知时，应用仍会消耗 CPU 资源。

iPhone（5、5S 和 6）和 iPad（3、4 和 Air）的处理器是双核或三核的。你可以在表 3-1 中查看完整的列表。Geekbench 的打分反映了这些常见主流 iOS 设备处理器的相对计算速度。

表3-1：iOS设备与处理器

设备	处理器	核心数	地址长度	CPU时钟	单核Geekbench	多核Geekbench[1]
iPhone 5	A6	2	32 bit	1.3 GHz	569	950
iPhone 5S	A7	2	64 bit	1.3~1.4 GHz	1400	2524
iPhone 5C	A6	2	32 bit	1.3 GHz	689	1243
iPhone 6	A8	2	64 bit	1.4 Ghz	1621	2899
iPhone 6 Plus	A8	2	64 bit	1.4 Ghz	1619	2902
iPhone 6S	A9	2	64 bit	1.8 Ghz	2487	4327
iPhone 6S Plus	A9	2	64 bit	1.8 Ghz	2478	4330
iPad 3	A5X	2	32 bit	1 Ghz	261	495
iPad 4	A6X	2	32 bit	1.4 Ghz	781	1422
iPad Air	A7	2	64 bit	1.4 Ghz	1462	2636
iPad Air 2	A8X	3	64 bit	1.5 Ghz	1815	4502

应用计算得越多，消耗的电量就越多。在完成相同的基本操作时，老一代的设备会消耗更多的电量。计算量的消耗取决于不同的因素。

- 对数据的处理（例如，对文本进行格式化）。
- 待处理的数据大小——更大的显示屏允许软件在单个视图中展示更多的信息，但这也意味着要处理更多的数据。
- 处理数据的算法和数据结构。
- 执行更新的次数，尤其是在数据更新后，触发应用的状态或 UI 进行更新（应用收到的推送通知也会导致数据更新，如果此时用户正在使用应用，你还需要更新 UI）。

没有单一规则可以减少设备中的执行次数。很多规则都取决于操作的本质。以下是一些可以在应用中投入使用的最佳实践。

- 针对不同的情况选择优化的算法

  例如，当你在排序时，如果列表少于 43 个实例，则插入排序优于归并排序，但实例多于 286 个时，应当使用快速排序。要优先使用双枢轴快速排序而不是传统的单枢轴快速排序。

- 如果应用从服务器接收数据，尽量减少需要在客户端进行的处理

  例如，如果一段文字需要在客户端进行渲染，尽可能在服务器将数据清理干净。

  我曾经做过一个项目，因为服务器的实现主要用于服务桌面用户，所以返回的文本中包含 HTML 标签。清理 HTML 标签的工作并没有放在客户端进行，而是放在了服务器端实现，从而减少了设备上的计算过程，降低了处理时间。

  我们在另一个项目中意识到，如果每次用户开启应用的数据差异较大，需要同步的记录开销也是相当高的。我们并没有用传统的方式，通过服务器下发更新的增量数据，而是将其设置成发送一个二进制的数据库，用于替换设备上已有的数据库。这不仅确保了对网络的优化使用，还节省了在本地设备上合并数据所需的计算。

注 1：https://browser.primatelabs.com/ios-benchmarks

- 优化静态编译（ahead-of-time，AOT）处理
  动态编译（just-in-time，JIT）处理的缺点在于它会强制用户等待操作完成。但是激进的 AOT 处理则会导致计算资源的浪费。需要根据应用和设备选择精确定量的 AOT 处理。

  例如，在 `UITableView` 中渲染一组记录时，在载入列表时处理全部的记录并不是明智的选择。基于单元格的高度，如果设备可以渲染 *N* 条记录，那么 3*N* 或 4*N* 则是一个理想的数据载入规模。类似地，如果用户快速滚动，则不应立即载入记录，而应推迟到滚动速度下降至某一阈值。精确的阈值应该由每个单元格的处理时间和单元格 UI 的复杂性来决定（例如，单元格中包含多张图片或视频）。

- 分析电量消耗
  测量目标用户的所有设备上的电量消耗（参见 3.7 节）。找到高能耗的区域并想办法降低能耗。

3.2　网络

智能的网络访问管理可以让应用响应得更快，并有助于延长电池寿命。在无法访问网络时，应当推迟后续的网络请求，直到网络连接恢复为止。

此外，应避免在没有连接 WiFi 的情况下进行高带宽消耗的操作，比如视频流。众所周知，蜂窝无线系统（LTE、4G、3G 等）对电量的消耗远大于 WiFi 信号。根源在于 LTE 设备基于多输入、多输出技术，使用多个并发信号以维护两端的 LTE 链接。类似地，所有的蜂窝数据连接都会定期扫描以寻找更强的信号。

因此，我们需要：

- 在进行任何网络操作之前，先检查合适的网络连接是否可用；
- 持续监视网络的可用性，并在连接状态发生变化时给予适当的反馈。

苹果公司提供了示例代码（http://apple.co/1Q3gRKL），以检查和监听网络状态的变化。如果你的项目使用了 CocoaPods，那么请使用 Tony Million 的 `Reachability`pod（https://github.com/tonymillion/Reachability）。

例 3-1 展示了在你的代码中添加一个简单的方法（`isAPIServerAvailable`），并在真正调用前使用该方法。

例 3-1　检查网络状态

```
//Helper API
-(BOOL)isAPIServerReachable {
  Reachability *r = [Reachability reachabilityWithHostname:@"api.yourdomain.com"];❶

  return r.isReachable;❷
}

//真正的网络操作
-(void)performNetworkOperation:(NSDictionary *)params
    completion:(void (^)(NSError *, id)) completion { ❸
```

```
if(!self.isAPIServerReachable) {
    [self enqueueRequest:params completion:completion]; ❹

    NSError *err = [[NSError alloc] initWithDomain:@"network"
        code:kErrorCodeNetworkUnreachable userInfo:nil];
    completion(err, nil); ❺
  } else {
    [self doNetworkOperation:params completion:completion]; ❻
  }
}
```

❶ 检查服务器域名是否可达。

❷ 可以随意地对 `isReachableViaWiFi` 或 `isReachableViaWWAN`（3G、4G、EDGE 等）方法进行优化改造。

❸ `completion` 回调方法提供了（指定操作的）`id` 类型的结果或 `NSError *` 类型的错误。

❹ 对操作进行排队。`queue` 方法的实现并没有在这里体现出来。

❺ `kErrorCodeNetworkUnreachable` 是在应用中定义的一个常量。

❻ 如果网络可用，立即触发请求。

类似地，为了实现第二步（监听网络状态并在网络可用时执行队列），你可以使用例 3-2 中的代码。

例 3-2　监控网络并执行队列

```
//HPNetworkOps.h
@property (nonatomic, readonly) BOOL isAPIServerReachable;

//HPNetworkOps.m

@property (nonatomic, strong) Reachability *reachability;
@property (nonatomic, strong) NSOperationQueue *networkOperationQueue;

-(id)init {
    if(self = [super init]) {
        self.reachability = [Reachability
                reachabilityWithHostname:@"api.yourdomain.com"];
        self.reachability.reachableOnWWAN = NO;

        self.networkOperationQueue = [[NSOperationQueue alloc] init];
        self.networkOperationQueue.maxConcurrentOperationCount = 1;

        [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(networkStatusChanged:)
            name:kReachabilityChangedNotification object:nil];
    }
    return self;
}

-(void)networkStatusChanged:(Reachability *)reachability {
    if(!reachability.isReachableViaWiFi) {
        self.networkOperationQueue.suspended = YES;
    } else {
        self.networkOperationQueue.suspended = NO;
```

```
        }
    }

    -(BOOL)isAPIServerReachable {
      return self.reachability.isReachableWiFi;
    }

    -(void)performNetworkOperation:(NSDictionary *)params
        completion:(void (^)(NSError *, id)) completion {
        [self enqueueRequest:params completion:completion];
    }

    -(void)enqueueRequest:(NSDictionary *)params
        completion:(void (^)(NSError *, id)) completion
    {
        NSURLRequest *req = ...;
        AFHTTPRequestOperation *op =
            [[AFHTTPRequestOperation alloc] initWithRequest:req];

        [op setCompletionBlockWithSuccess:^(AFHTTPRequestOperation *op, id res){
            completion(nil, res);
        } failure:^(AFHTTPRequestOperation *operation, NSError *error) {
            completion(error, nil);
        }];

        [self.networkOperationQueue addOperation:op]
    }
```

以下是对例 3-2 的代码分析。

- 类 `HPNetworkOps` 有一个 `isAPIServerReachable` 属性，该属性可以用于检测网络是否可用。取决于应用、应用的状态或者具体的任务，你可以使用这个标记来判断是继续处理，还是使用一个平视显示器来阻碍应用的交互。
- 这个类具有私有属性 `reachability` 和 `networkOperationQueue`。
    - `reachability` 属性用于监听状态。它仅用于跟踪 WiFi 网络的变化。
    - `networkOperationQueue` 保留了队列中的操作。该队列一次只允许执行一个操作。
- 根据网络的可用情况，通知的接收者（`networkStatusChanged`）挂起或恢复队列。
- 更新了 `performNetworkOperation` 的实现，从而总是将网络操作送入队列中。
- `enqueueRequest:completion:` 方法对网络操作进行排队。

  这个例子中的 `AFHTTPRequestOperation` 来源于 `AFNetworking`pod（https://github.com/AFNetworking/AFNetworking）。你可以自由选择其他的操作或创建自己的操作。

在这段代码中，我们用 `NSOperationQueue` 进行了演示。你可能还需要一个更加复杂的队列来实现额外的控制。更极端一点的话，你可能需要将暂停的网络操作保存起来，以便后续网络恢复可用时再与服务器同步更新。

注意，`NSOperationQueue` 不会暂停或挂起任何执行中的操作。一个挂起的队列仅仅意味着后续操作在其恢复之前不会被执行。正如苹果公司的开发者文档所描述的那样。

操作只有完成后才会从队列中移除。然而，为了完成执行，必须先启动操作。因为挂起的队列不会启动任何新的操作，所以它也不会移除任何正在排队且未被执行的操作（包括那些已经取消的操作）。

使用基于队列的网络请求以避免服务器被多个同时发起的请求所轰炸。至少使用两个队列：一个用于通常不是很关键的大量图片下载；另一个用于关键数据的请求。参见 4.4 节以了解使用操作的详细信息。

此外，作为一位有情怀的工程师，你还需要更新网络活动指示符——将操作加入队列时打开它，收到网络响应后将其关闭。这里需要使用定义在 `UIApplication` 类中的 `setNetworkActivityIndicatorVisible` 方法。

3.3 定位管理器和GPS

了解定位服务包括 GPS（或 GLONASS）和 WiFi 硬件这一点很重要，同时要知道定位服务需要大量的电量。

使用 GPS 计算坐标需要确定两点信息。

- 时间锁
  每个 GPS 卫星每毫秒广播唯一一个 1023 位随机数，因而数据传播速率是 1.024Mbit/s。GPS 的接收芯片必须正确地与卫星的时间锁槽对齐。

- 频率锁
  GPS 接收器必须计算由接收器与卫星的相对运动导致的多普勒偏移带来的信号误差。

通常情况下，锁定一颗卫星至少需要 30 秒。必须锁定接收范围内的所有卫星。确定的卫星越多，取得的定位坐标就越精确。

计算坐标会不断地使用 CPU 和 GPS 的硬件资源，因此它们会迅速地消耗电池电量。

GPS 精密码

卫星的精度码更广为人知的名字是 P-code，实际长度为 6.1871×10^{12} 位，大约有 720GB。它以 10.23Mbit/s 的速度传输，每周重复一次。

更有趣的是，这个 P-code 只是一个主 P-code 的子集。主 P-code 的长度接近 2.34×10^{14} 位，大约有 26.7TB。

现在你已经了解了 GPS 锁定过程的复杂本质，再怎么强调要对其小心使用都不为过，尤其是在你的应用重度依赖地图的情况下。

接下来我们将讨论可以最小化电量使用的最佳实践（你的客户一定会因为这项优化而格外满意）。

例 3-3 展示了初始化 `CLLocationManager` 并高效接收地理位置更新的典型代码。

例 3-3　使用定位管理器

```
//HPLocationViewController.h
@interface HPLocationViewController : UIViewController <CLLocationManagerDelegate>

@property (nonatomic, strong) CLLocationManager *manager;

@end

//HPLocationViewController.m
@implementation HPLocationViewController

-(void)viewDidLoad
{
    self.manager = [[CLLocationManager alloc] init]; ❶
    self.manager.delegate = self;
}

-(IBAction)enableLocationButtonTapped:(id)sender
{
    self.manager.distanceFilter = kCLDistanceFilterNone; ❷
        self.manager.desiredAccuracy = kCLLocationAccuracyBest; ❸

    if(isIOS8()) {❹
        [self.manager requestWhenInUseAuthorization]; ❺
    }

    [self.manager startUpdatingLocation];
}

-(void)locationManager:(CLLocationManager *)manager
    didUpdateLocations:(NSArray *)locations
{
    CLLocation *loc = [locations lastObject];
    //使用定位信息
}

@end
```

❶ 这里没有使用依赖注入，但此处的 `manager` 被视图控制器所持有、管理和配置。
❷ 初始化管理器，观察所有距离的变化。
❸ 按照最大精度初始化管理器。
❹ 为了简化代码，这里省略了 `isIOS8` 辅助方法。当应用在 iOS 8 及更新版本的系统上运行时，它会返回 `true`。
❺ 这是 iOS 8 特定的一个 API，用于在应用活动时申请使用定位服务。

3.3.1　最佳的初始化

正如例 3-3 所示，在调用 `startUpdatingLocation` 方法时，两个参数起着非常重要的作用。

- `distanceFilter`
  只要设备的移动超过了最小距离，距离过滤器就会导致管理器对委托对象的 `locationManager:didUpdateLocations:` 事件通知发生变化。该距离使用公制单位（米）。

这并不会有助于减少 GPS 接收器的使用，但会影响应用的处理速度，从而直接减少 CPU 的使用。

- `desiredAccuracy`
  精度参数的使用直接影响了使用天线的个数，进而影响了对电池的消耗。精度级别的选取取决于应用的具体用途。按照降序排列，精度由以下常量定义。

  - `kCLLocationAccuracyBestForNavigation`
    用于导航的最佳精度级别。
  - `kCLLocationAccuracyBest`
    设备可能达到的最佳精度级别。
  - `kCLLocationAccuracyNearestTenMeters`
    精度接近 10 米。如果对用户所走的每一米并不感兴趣，不妨使用这个值（例如，可在测量大块距离时使用）。
  - `kCLLocationAccuracyHundredMeters`
    精度接近 100 米（在计算距离时，这个值需要乘以 100 米）。
  - `kCLLocationAccuracyKilometer`
    精度在千米范围。这在粗略测量两个距离数百千米的兴趣点时非常有用（例如，如果计算从旧金山的家到阿纳海姆的迪斯尼乐园的距离，这个精度就已足够）。
  - `kCLLocationAccuracyThreeKilometers`
    精度在 3 千米范围。在距离真的很远时使用这个值（如果计算位于英国伦敦的家到印度泰姬陵的距离，这个精度也就足够了）。

> 距离过滤器只是软件层面的过滤器，而精度级别会影响物理天线的使用。
> 当委托的回调方法 `locationManager:didUpdateLocations:` 被调用时，使用距离范围更广的过渡器只会影响间隔。另一方面，更高的精度级别意味着更多的活动天线，这会消耗更多的能量。

3.3.2 关闭无关紧要的特性

判断何时需要跟踪位置的变化。在需要跟踪时调用 `startUpdatingLocation` 方法，无需跟踪时调用 `stopUpdatingLocation` 方法。

假设用户需要用一个消息类的应用与朋友分享位置。如果该应用只是发送城市的名称，则只需要一次性地获取位置信息，然后就可以通过调用 `stopUpdatingLocation` 关闭位置跟踪。在一定的时间间隔后可以再次开启定位。你可以设置固定的间隔（如 60 秒或 5 分钟），也可以动态地计算时间间隔（例如，根据之前获取的坐标和速度，估算穿过城市的时间上限）。

当应用在后台运行或用户没有与别人聊天时，也应该关闭位置跟踪。这也就是说，浏览媒体库、查看朋友列表或调整应用设置时，都应该关闭位置跟踪。

向终端用户提供关闭非必要功能的选项是一个更好的解决方案。例如，Waze 应用提供了一个选项以关闭应用中的所有活动（如图 3-1 所示）。

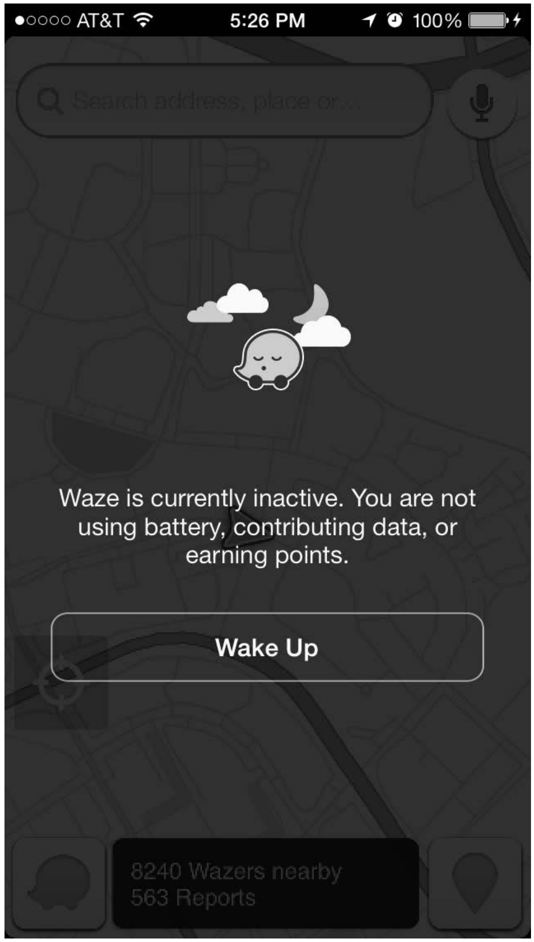


图 3-1：关闭了位置跟踪的 Waze 应用

3.3.3　只在必要时使用网络

为了提高电量的使用效率，iOS 总是尽可能地保持无线网络关闭。当应用需要建立网络连接时，iOS 会利用这个机会向后台应用分享网络会话，以便一些低优先级的事件能够被处理，如推送通知、收取电子邮件等。

关键在于每当应用建立网络连接时，网络硬件都会在连接完成后多维持几秒的活动时间。每次集中的网络通信都会消耗大量的电量。

要想减轻这个问题带来的危害，你的软件需要有所保留地使用网络。应该定期集中短暂地使用网络，而不是持续地保持着活动的数据流。只有这样，网络硬件才有机会被关闭。

3.3.4 后台定位服务

CLLocationManager 提供了一个替代的方法来监听位置的更新。startMonitoringSignificantLocationChanges 可以帮助你在更远的距离跟踪运动。精确的值由内部决定，且与 distanceFilter 无关。

使用这一模式可以在应用进入后台后继续跟踪运动。（除非应用是导航类应用，且你想在锁屏期间也获得很好的细节。）典型的做法是在应用进入后台时执行 startMonitoringSignificantLocationChanges 方法，而当应用回到前台时执行 startUpdatingLocation。你可以在自己的应用中使用例 3-4 中的示例代码。

例 3-4　监听与重大变化监听

```
//应用委托
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    [self.locationManager stopUpdatingLocation];
    [self.locationManager startMonitoringSignificantLocationChanges];
}

- (void)willEnterForeground:(UIApplication *)application
{
    [self.locationManager stopMonitoringSignificantLocationChanges];
    [self.locationManager startUpdatingLocation];
}
```

在 iOS 8 系统中的后台使用定位

当应用位于 iOS 8 中的后台时，你需要显式地申请权限来使用定位管理器。为了获取用户的授权，必须完成以下步骤。

(1) 更新应用的 Info.plist 文件，添加 String 类型的 NSLocationAlwaysUsageDescription 条目。相应的值是应用进入后台或前台时向用户申请权限时显示的消息。当应用只在前台时，使用 NSLocationWhenInUseUsageDescription 获取使用位置的权限。

(2) 调用 requestAlwaysAuthorization 方法申请前台和后台权限（requestWhenInUseAuthorization 方法仅适用于前台使用）。



与 iOS 7 不同，startUpdatingLocation 方法在 iOS 8 中不再申请用户权限以便使用定位数据。你必须使用 requestWhenInUseAuthorization 或 requestAlwaysAuthorization。

3.3.5 NSTimer、NSThread和定位服务

当应用位于后台时，任何定时器或线程都会挂起。但如果你在应用位于后台状态时申请了定位，那么应用会在每次收到更新后被短暂唤醒。在此期间，线程和计时器都会被唤醒。

可怕之处在于，如果你在这段时间做了任何网络操作，则会启动所有相关的天线（如 WiFi 和 LTE/4G/3G）。

想要控制这种状况往往非常棘手。最佳的选择是使用 NSURLSession 类。我们会在 7.1.5 节讨论“网络 API”。

3.3.6 在应用关闭后重启

同样重要的是，在其他应用需要更多资源时，后台的应用可能会被关闭。在这种情况下，一旦发生位置变化，应用会被重启，因而需要重新初始化监听过程。若出现这种情况，application:didFinishLaunchingWithOptions: 方法会收到键值为 UIApplicationLaunchOptionsLocationKey 的条目。相关代码见例 3-5。

例 3-5　在应用关闭后重新初始化监听

```
-(void)application:(UIApplication *)app
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    if(launchOptions[UIApplicationLaunchOptionsLocationKey]) { ❶
        [self.manager startMonitoringSignificantLocationChanges]; ❷
    }
}
```

❶ 因缺乏资源而关闭应用后，检测应用是否因为位置变化而被重启了。

❷ 如果是这样的话，开始监听位置的变化。如果不是，则在后续适合的时机开始监听。

3.4 屏幕

屏幕非常耗电。屏幕越大就越费电。当然，如果你的应用在前台运行且与用户进行交互，则势必会使用屏幕并消耗电量。

然而，仍然有一些方案可以优化屏幕的使用。

3.4.1 动画

明智地使用动画是一个被遗弃的概念。尽管如此，我们讨论这个问题是为了表述的完整性。

你可以遵守一个简单的规则：当应用在前台时使用动画，一旦应用进入后台则立即暂停动画。通常来说，你可以通过监听 UIApplicationWillResignActiveNotification 或 UIApplicationDidEnterBackgroundNotification 的通知事件来暂停或停止动画，也可以通过监听 UIApplicationDidBecomeActiveNotification 的通知事件来恢复动画。

3.4.2 视频播放

在视频播放期间，最好强制保持屏幕常亮。可以使用 UIApplication 对象的 idleTimerDisabled 属性来实现这个目的。一旦设置为 YES，它会阻止屏幕休眠，从而实现常亮。

与动画类似，你可以通过响应应用的通知来释放和获取锁。

3.4.3 多屏幕

使用屏幕比休眠锁或暂停 / 恢复动画要复杂很多。

如果设备连接了外部显示设备（使用 AirPlay 或 HDMI 连接器），那么会发生什么呢？大部分应用通常表现为操作系统的默认行为，即以镜像方式向外部显示设备投影。

但实际上还可以做得更多。如果正在播放电影或运行动画，你可以将它们从设备的屏幕挪到外部屏幕，而只在设备的屏幕上保留最基本的控制。这样可以减少设备上的屏幕更新，进而延长电池寿命。苹果公司的开发者网站提供了使用外部屏幕（http://apple.co/1jauUnu）的简单示例。使用数据线将 iPhone 或 iPad 连接到汽车的屏幕或使用 AirPlay 将它们连接到 AppleTV，并非是罕见的事情。

处理这一场景的典型代码会涉及以下步骤。

(1) 在启动期间检测屏幕的数量。
　如果屏幕数量大于 1，则进行切换。

(2) 监听屏幕在连接和断开时的通知。
　如果有新的屏幕加入，则进行切换。
　如果所有的外部屏幕都被移除，则恢复到默认显示。

例 3-6 展示了如何通过多个屏幕获取效益。

例 3-6　使用多个屏幕

```
//HPMultiScreenViewController.m
@interface HPMultiScreenViewController ()

@property (nonatomic, strong) UIWindow *secondWindow;

@end

@implementation HPMultiScreenViewController

-(void)viewDidLoad
{
    [super viewDidLoad];
    [self registerNotifications];
}

-(void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    [self updateScreens];
}

-(void)viewDidDisappear:(BOOL)animated
{
    [super viewDidDisappear:animated];
    [self disconnectFromScreen];
}
```

```
-(void)disconnectFromScreen
{
    if(self.secondWindow != nil) {
        //断开链接并准备释放内存
        self.secondWindow.rootViewController = nil;
        self.secondWindow.hidden = YES;
        self.secondWindow = nil;
    }
}

-(void)updateScreens
{
    NSArray *screens = [UIScreen screens];
    if(screens.count > 1) {
        UIScreen *secondScreen = (UIScreen *)[screens objectAtIndex:1];
        CGRect rect = secondScreen.bounds;
        if(self.secondWindow == nil) {
            self.secondWindow = [[UIWindow alloc] initWithFrame:rect];
            self.secondWindow.screen = secondScreen;

            HPScreen2ViewController *svc = [[HPScreen2ViewController alloc] init];
            //设置svc的其他属性以完整地对它进行初始化
            svc.parent = self;
            self.secondWindow.rootViewController = svc;
        }
        self.secondWindow.hidden = NO;
    } else {
        [self disconnectFromScreen];
    }
}

-(void)dealloc
{
    [self unregisterNotifications];
}

-(void)screensChanged:(NSNotification *)notification
{
    [self updateScreens];
}

-(void)registerNotifications
{
    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    [nc addObserver:self
            selector:@selector(screensChanged:)
        name:UIScreenDidConnectNotification object:nil];
        [nc addObserver:self
            selector:@selector(screensChanged:)
        name:UIScreenDidDisconnectNotification object:nil];
}

-(void)unregisterNotifications
{
```

```
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}

@end
```

在例 3-6 中，HPMultiScreenViewController 是包含视频播放或动画 UI 的视图控制器。

在此示例中，我们使用了另外一个辅助视图控制器——HPScreen2ViewController。它会和父视图控制器通信，并根据用户的交互发送适当的消息。以下是对每个方法的详细介绍。

- viewDidLoad
  因为这个方法在视图控制器的生命周期中会且仅会被调用一次，所以它就成为了注册 UIScreenDidConnectNotification（屏幕连接）通知和 UIScreenDidDisconnectNotification（屏幕断开）通知的最佳位置。
- 每当有新的屏幕加入或有屏幕移除时，我们都会调用 screensChanged: 方法，在这里更新 UI。
- viewDidAppear:
  由于视图可以显示或消失多次，或者更具体来说，用户可以进入或离开视图多次，我们可以用这个方法更新屏幕。
- 用户首次进入视图控制器时，UI 会进行调整，查看可用的屏幕数量。类似地，如果用户离开这个视图控制器，转而进入其他视图控制器，然后再返回，此时屏幕数量可能已经发生变化。因此，它们可能需要进行调整。
- viewDidDisappear:
  当用户离开这个视图控制器时，你可能也想在另一个屏幕上更新 UI。使用这个方法就能实现这一操作。

  在上述示例中，我们从屏幕中移除了 secondWindow（通过 disconnectFromScreen 方法）。

  在一个更加复杂的应用中，你可能需要支持在外部屏幕上播放视频的同时允许用户进行复杂的操作，例如，在设备屏幕上重新排列播放列表或搜索媒体。

  图 3-2 展示了使用类似应用时的模拟 UI。

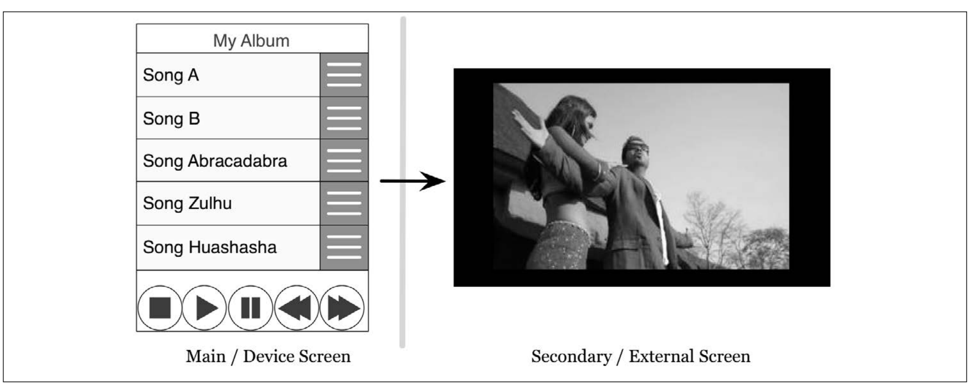


图 3-2：多屏 UI

- `disconnectFromScreen`

  调用这个方法将 `secondWindow` 从屏幕上移除。

- `updateScreens`

  这是真正的魔法产生的地方，尽管示例出于演示目的极力保持简洁。

  在这个方法中，我们检查了屏幕的数量，如果数量大于 1，我们就将新的窗口与第二屏连接。在实际使用时，你可以扫描遍历所有的屏幕，并决定对哪一个屏幕进行操作——最简单的操作就是将 UI 复制到所有屏幕上。

  如果屏幕数是 1，则调用 `disconnectFromScreen` 方法。

- `dealloc`

  当视图控制器即将被销毁时调用一次，我们用这个方法解除对屏幕通知的注册。

- `screensChanged:`

  当应用获取屏幕断开的通知时，该方法调用 `updateScreens`。

- `registerNotifications`

  这个方法添加了 `UIScreenDidConnectNotification` 和 `UIScreenDidDisconnectNotification` 通知的观察者。

- `unregisterNotifications`

  这个方法移除了观察者。

在实际的应用中，`HPScreen2ViewController` 将由与用户交互以实现控制的 UI 组成，如电影播放器。你可能也会在不同的屏幕间切换控制器。例 3-7 展示了如何实现切换。

例 3-7　在屏幕之间交换 UI

```
-(void)swapScreens(UIWindow *)currentWindow newWindow:(UIWindow *)newWindow
{
    NSArray *screens = [UIScreen screens];

    UIScreen *deviceScreen = [screens objectAdIndex:0];
    UIScreen *extScreen = [screens objectAdIndex:1];

    //你可以视情况合理地设置边界
    currentWindow.screen = extScreen;
    newWindow.screen = deviceScreen;
}
```

> 在一个屏幕上显示却在另一个屏幕上控制似乎显得有些笨拙，但这样可以实现不中断的显示。此外，如果控制是标准按钮（如播放、暂停、恢复、停止等），实际体验并不会变差。
>
> 当然，不要在交互游戏中使用这种方法，因为交互游戏需要在屏幕上进行操作。如果你这么做了，用户会十分不满，因为他们无法在一个空白的屏幕上动态地操控游戏。

3.5　其他硬件

当应用进入后台时，应该释放对这些硬件的锁定：

- 蓝牙
- 相机
- 扬声器，除非应用是音乐类的
- 麦克风

我们并不会在这里讨论这些硬件的特性，但是基本规则是一致的——只有当应用处于前台时才与这些硬件进行交互，应用处于后台时应停止交互。

扬声器和无线蓝牙可能是例外。如果你正在开发音乐、收音机或其他的音频类应用，则需要在应用进入后台后继续使用扬声器。不要让屏幕仅仅为音频播放的目的而保持常亮。类似地，若应用还有未完成的数据传输，则需要在应用进入后台后持续使用无线蓝牙，例如，与其他设备传输文件。

3.6　电池电量与代码感知

一个智能的应用会考虑到电池的电量和自身的状态，从而决定是否要真正执行资源密集消耗型的操作。另外一个有价值的点是对充电的判断，确定设备是否处于充电状态。

使用 `UIDevice` 实例可以获取 `batteryLevel` 和 `batteryState`（充电状态）。你可以将例 3-8 中的代码直接用于自己的应用。`ceedWithMinLevel:` 方法传入执行特定操作需要的最低电量级别。该级别是浮点数，范围在 0~100（100 表示电池完全充满）。

例 3-8　使用电量级别和充电状态进行条件处理

```
-(BOOL)shouldProceedWithMinLevel:(NSUInteger)minLevel
{
    UIDevice *device = [UIDevice currentDevice];
    device.batteryMonitoringEnabled = YES;

    UIDeviceBatteryState state = device.batteryState;
    if(state == UIDeviceBatteryStateCharging ||
        state == UIDeviceBatteryStateFull) { ❶
        return YES;
    }

    NSUInteger batteryLevel = (NSUInteger) (device.batteryLevel * 100); ❷
    if(batteryLevel >= minLevel) {
        return YES;
    }
    return NO;
}
```

❶ 在充电或电池已经充满的情况下，任何操作都可以执行。

❷ `UIDevice` 返回的 `batteryLevel` 的范围在 0.00~1.00。

类似地，你还可以得到应用对 CPU 的利用率。在应用运行时这可能并不是一个特别重要

的信息，但是出于完整性的目的，我们在例 3-9 中给出了这部分代码。

例 3-9　应用对 CPU 的使用率

```
-(float)appCPUUsage {
  kern_return_t kr;
        task_info_data_t info;
        mach_msg_type_number_t infoCount = TASK_INFO_MAX;

        kr = task_info(mach_task_self(), TASK_BASIC_INFO,
                      (task_info_t)info, &infoCount);
        if (kr != KERN_SUCCESS) {
                return -1;
        }

        thread_array_t         thread_list;
        mach_msg_type_number_t thread_count;
        thread_info_data_t     thinfo;
        mach_msg_type_number_t thread_info_count;
        thread_basic_info_t    basic_info_th;

 kr = task_threads(mach_task_self(), &thread_list, &thread_count);
        if (kr != KERN_SUCCESS) {
                return -1;
        }

        float tot_cpu = 0;
        int j;

        for (j = 0; j < thread_count; j++) {
                thread_info_count = THREAD_INFO_MAX;
                kr = thread_info(thread_list[j], THREAD_BASIC_INFO,
                                (thread_info_t)thinfo, &thread_info_count);
                if (kr != KERN_SUCCESS) {
                        return -1;
                }

               basic_info_th = (thread_basic_info_t)thinfo;

               if (!(basic_info_th->flags & TH_FLAGS_IDLE)) {
                       tot_cpu += basic_info_th->cpu_usage /
                               (float)TH_USAGE_SCALE * 100.0;
               }
          }

          vm_deallocate(mach_task_self(), (vm_offset_t)thread_list,
                 thread_count * sizeof(thread_t));
          return tot_cpu;
}
```

当剩余电量较低时提示用户，并请求用户授权执行电源密集型的操作——当然，只在用户同意的前提下执行。

总是用一个指示符显示长时间任务的进度，包括设备上即将完成的计算或者只是下载一些内容。向用户提供完成进度的估算，以帮助他们决定是否需要为设备充电。

3.7 分析电量使用

在开发期间使用 Xcode Instruments 跟踪 CPU 的使用。我们将在第 11 章对该工具进行深入探讨。现在我们对其中的活动监视器（参见 11.2.2 节）模板更感兴趣。它很好地提供了测量电量消耗的能力，因为 CPU 消耗了最多的电量。

要想得到准确量化的应用的电量消耗，你可以使用 Monsoon Solutions 的电源监控器（Power Monitor，https://www.msoon.com/LabEquipment/PowerMonitor）。使用这个工具的步骤如下。

(1) 拆开 iOS 设备的外壳，找到电池后面的电源针脚。
(2) 连接电源监控器的设备针脚。
(3) 运行应用。
(4) 测量电量消耗。

图 3-3 展示了与 iPhone 的电池针脚连接的电源监控器工具。

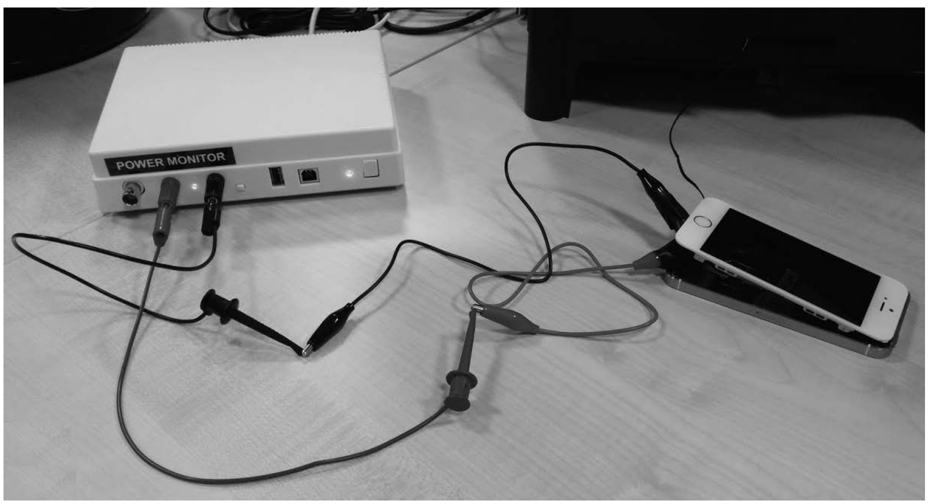


图 3-3：电源监控器与 iPhone 5S 进行连接（图片摘自 Bottle of Code）

电源监控器软件可以跟踪电源随时间流逝的使用情况。数据以图表的形式展示出来了，如图 3-4 所示。

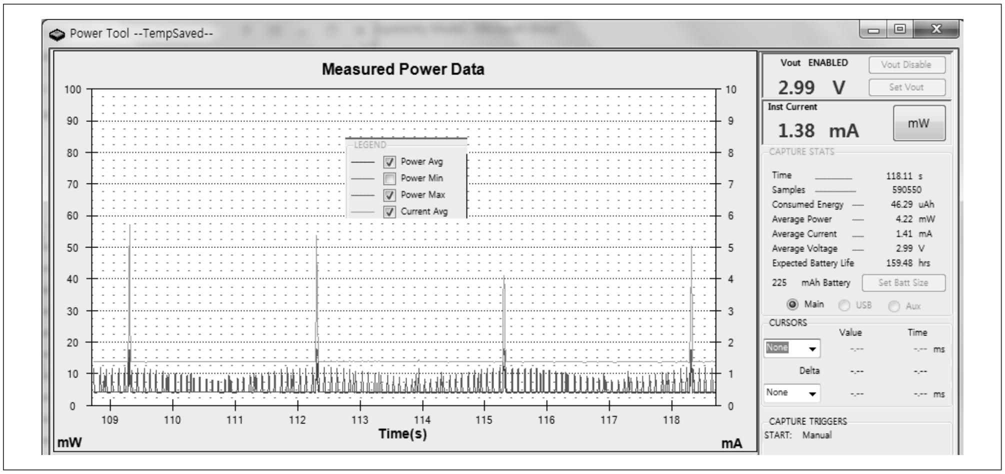


图 3-4：电源监控器软件

3.8 最佳实践

以下的最佳实践可以确保对电量的谨慎使用。遵循以下要点，应用可以实现对电量的高效使用。

- 最小化硬件使用。换句话说，尽可能晚地与硬件打交道，并且一旦完成任务立即结束使用。
- 在进行密集型任务前，检查电池电量和充电状态。
- 在电量低时，提示用户是否确定要执行任务，并在用户同意后再执行。
- 或提供设置的选项，允许用户定义电量的阈值，以便在执行密集型操作前提示用户。

例 3-10 的示例代码展示了设定电量的阈值以提示用户。阈值的配置见图 3-5。

例 3-10　如果电量低，在执行密集型操作之前提示用户

```
-(IBAction)onIntensiveOperationButtonClick:(id)sender {

 NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
        BOOL prompt = [defaults boolForKey:@"promptForBattery"];
        int minLevel = [defaults integerForKey:@"minBatteryLevel"];

        BOOL canAutoProceed = [self shouldProceedWithMinLevel:minLevel];
        if(canAutoProceed) {
                [self executeIntensiveOperation];
        } else {
  if(prompt) {
   UIAlertView *view = [[UIAlertView alloc] initWithTitle:@"Proceed"
                       message:@"Battery level below minimum required. Proceed?"
                              delegate:self cancelButtonTitle:@"No"
    otherButtonTitles:@"Yes", nil];
   [view show];
  } else {
                      [self queueIntensiveOperation];
```

```
                }
            }
    }

    - (void)alertView:(UIAlertView *)alertView
      clickedButtonAtIndex:(NSInteger)buttonIndex {

      if(buttonIndex == 0) {
              [self queueIntensiveOperation];
            } else {
                    [self executeIntensiveOperation];
            }
    }
```

可以用以下方式解读例 3-10 中的代码。

- 点击按钮（或任何其他逻辑）就会执行 `onIntensiveOperationButtonClick:` 方法。假设该方法就会触发密集型的操作。
- 设置由两个条目组成：`promptForBattery`（应用设置中的拨动开关，表明是否要在低电量时给予提示）和 `miniBatteryLevel`（区间为 0~100 的一个滑块，表明了最低电量——在此示例中，用户可以自行调整）。应用设置的界面如图 3-5 所示。

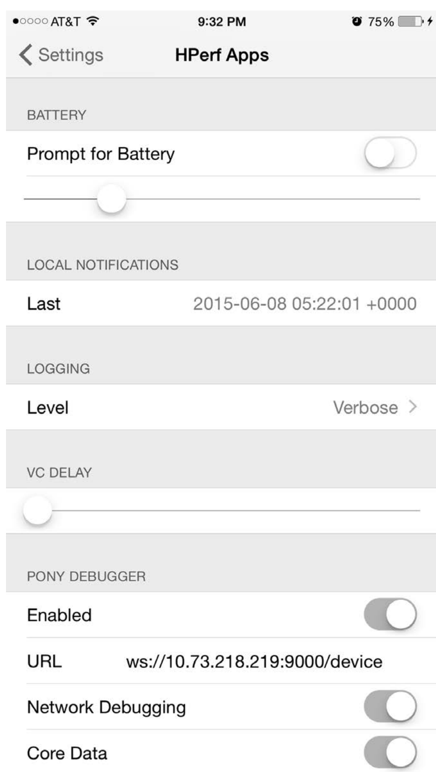


图 3-5：电量水平的阈值和提示选项在应用中的设置

在实际的应用中，应用的开发人员通常根据操作的复杂性和密集性对阈值进行预设。不同的密集型操作可能会有不同的最低电量需求。

- 在实际执行密集操作之前，检查当前电量是否足够，或者手机是否正在充电。这就是我判断是否可以进行后续处理的逻辑，如例 3-8 所示。你可以有自己的定制——最低电量和充电状态。
- 如果条件具备，则立即执行（这里需要在类中调用类似 `executeIntensiveOperation` 的方法）。
- 否则的话，如果用户设置了 `promptForBattery`，他会被提示。
  若用户没有选择被提示，我们将密集操作放入队列，以便后续执行（这里在类中调用类似 `queueIntensiveOperation` 的方法）。
- 提示之后，如果用户选择 `Ok`，那么我们调用 `executeIntensiveOperation`，否则调用 `queueIntensiveOperation`。

3.9 小结

因为用户总会随身携带移动设备，所以编写省电的代码显得格外重要，毕竟移动设备的充电接口并非随处可见，而且也不是所有的用户都会随身携带移动电源。

在无法降低任务复杂性（例如，处理图片或绘制图表）时，提供一个对电池电量保持敏感的方案并在适当的时机提示用户，会让用户感觉良好，并因此欣赏你的应用。

在下一章中，我们将讨论并行执行多个任务的方案和最佳实践。本章中优化内存、降低能耗的内容将为我们进一步讨论的话题提供素材。

第 4 章

并发编程

iOS 设备有两或三个 CPU 核心（见表 3-1）。这意味着，即使应用的主线程（UI 线程）正忙于更新屏幕，应用仍然可以在后台进行更多计算，而无需任何上下文的切换。

在本章中，我们将探索诸多充分利用现有 CPU 核心的方案，并学习如何通过并发编程优化性能。我们将讨论以下主题：

- 创建和管理线程
- 多线程优化技术（Grand Central Dispatch，GCD）概述
- 操作和队列

我们将讨论编写线程安全的高性能代码的技术和最佳实践。

4.1 线程

*线程*是运行时执行的一组指令序列。

每个进程至少应包含一个线程。在 iOS 中，进程启动时的主要线程通常被称作*主线程*。所有的 UI 元素都需要在主线程中创建和管理。与用户交互相关的所有中断最终都会分发到 UI 线程，处理代码会在这些地方执行——`IBAction` 方法的代码都会在主线程中执行。

Cocoa 编程不允许其他线程更新 UI 元素。这意味着，无论何时应用在后台线程执行了耗时操作，比如网络或其他处理，代码都必须将上下文切换到主线程再更新 UI——例如，进度条指示任务进度或标签展示处理结果。

4.2 线程开销

虽然应用有多个线程看起来非常赞，但每个线程都有一定的开销，从而影响到应用的性能。线程不仅仅有创建时的时间开销，还会消耗内核的内存，即应用的内存空间。[1]

4.2.1 内核数据结构

每个线程大约消耗 1KB 的内核内存空间。这块内存用于存储与线程有关的数据结构和属性。这块内存是联动内存（wired memory），无法被分页。

4.2.2 栈空间

主线程的栈空间大小为 1M，而且无法修改。所有的二级线程默认分配 512KB 的栈空间。注意，完整的栈并不会立即被创建出来。实际的栈空间大小会随着使用而增长。因此，即使主线程有 1MB 的栈空间，某个时间点的实际栈空间很可能要小很多。

在线程启动前，栈空间的大小可以被改变。栈空间的最小值是 16KB，而且其数值必须是 4KB 的倍数。例 4-1 中的示例代码展示了如何在启动线程前配置栈大小。

例 4-1　修改栈空间

```
+(NSThread *)createThreadWithTarget:(id)target selector:(SEL)selector
            object:(id)argument stackSize:(NSUInteger)size {

    if( (size % 4096) != 0) {
        return nil;
    }
    NSThread *t = [[NSThread alloc] initWithTarget:target
        selector:selector object:argument];
    t.stackSize = size;

    return t;
}
```

4.2.3 创建耗时

我们在 iPhone 6 Plus iOS 8.4 上进行了一项快速测试，展示了线程创建的耗时（不包含启动时间），其区间范围在 4000~5000 微秒，即 4~5 毫秒。

创建线程后启动线程的耗时区间为 5~100 毫秒，平均大约在 29 毫秒。这是很大的时间开销，若在应用启动时开启多个线程，则尤为明显。

线程的启动时间之所以如此之长，是因为多次的上下文切换所带来的开销。

出于简洁的目的，我们省略了计算的代码。要想了解细节，你可以参考 GitHub（https://github.com/gvaish/hpios/blob/master/src/ViewControllers/HPChapter05ViewController.m）中的 `computeThreadCreationTime` 方法。图 4-1 展示了这段代码的输出。

注 1：iOS Developer Library, “Thread Costs” (https://sites.google.com/site/appleiotemp//1EukJhy).

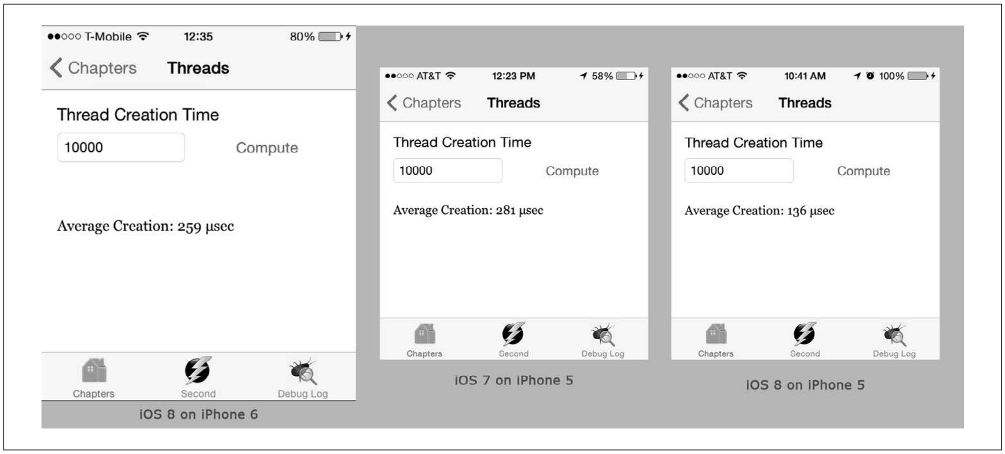


图 4-1：创建线程的耗时

4.3 GCD

GCD API（https://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/index.html）由核心语言特性、运行时库以及对执行并行代码的系统增强所组成。

我们不打算介绍使用 GCD 的基础知识，因为那不是本书的目的。你应该已经有一些使用 GCD 的经历，若需要复习 GCD 的基础知识，可以参阅 Ray Wenderlich 的“iOS 系统中的多线程和 GCD 的初学者教程”（https://www.raywenderlich.com/4295/multithreading-and-grand-central-dispatch-on-ios-for-beginners-tutorial）。

然而，出于完整性的考虑，我们快速看一下 GCD 提供的功能列表。

- 任务或分发队列，允许主线程中的执行、并行执行和串行执行。
- 分发组，实现对一组任务执行情况的跟踪，而与这些任务所基于的队列无关。
- 信号量。
- 屏障，允许在并行分发队列中创建同步的点。
- 分发对象和管理源，实现更为底层的管理和监控。
- 异步 I/O，使用文件描述符或管道。

GCD 同样解决了线程的创建与管理。它帮助我们跟踪应用中线程的总数，且不会造成任何的泄漏。

大多数情况下，应用单独使用 GCD 就可以很好地工作，但仍有特定的情况需要考虑使用 NSThread 或 NSOperationQueue。当应用中有多个长耗时的任务需要并行执行时，最好 . 对线程的创建过程加以控制。如果代码执行的时间过长，很有可能达到线程的限制 64 个，[2,3] 即 GCD 的线程池上限。

应该避免浪费地使用 dispatch_async 和 dispatch_sync，因为那会导致应用崩溃 [4]。虽然 64 个线程对移动应用来说是个很高的合理值，但不加控制的应用迟早会超出这个限制。

4.4 操作与队列

操作和操作队列是 iOS 编程中和任务管理有关的又一个重要概念。

NSOperation 封装了一个任务以及和任务相关的数据和代码，而 NSOperationQueue 以先入先出的顺序控制了一个或多个这类任务的执行。

NSOperation 和 NSOperationQueue 都提供控制线程个数的能力。可用 maxConcurrentOperationCount 属性控制队列的个数，也可以控制每个队列的线程个数。

在使用 NSThread（开发人员管理全部并发）和 GCD（OS 管理并发）之间存在两个选择。

以下是对 NSThread、NSOperationQueue 和 GCD API 的一个快速比较。

- GCD
    - 抽象程度最高。
    - 两种队列开箱即用：main 和 global。
    - 可以创建更多的队列（使用 dispatch_queue_create）。
    - 可以请求独占访问（使用 dispatch_barrier_sync 和 dispatch_barrier_async）。
    - 基于线程管理。
    - 硬性限制创建 64 个线程。
- NSOperationQueue
    - 无默认队列。
    - 应用管理自己创建的队列。
    - 队列是优先级队列。
    - 操作可以有不同的优先级（使用 queuePriority 属性）。

注 2：Stack Overflow, “Number of Threads Created by GCD?” (http://stackoverflow.com/questions/7213845/number-of-threads-created-by-gcd#0).

注 3：Stack Overflow, “Workaround on the Threads Limit in Grand Central Dispatch?” (http://stackoverflow.com/questions/15150308/workaround-on-the-threads-limit-in-grand-central-dispatch#0).

注 4：Stack Overflow, “GCD Dispatch Concurrent Queue Freeze with ‘Dispatch Thread Soft Limit Reached: 64’ in Crash Log” (http://stackoverflow.com/questions/14027824/gcd-dispatch-concurrent-queue-freeze-with-dispatch-thread-soft-limit-reached-6#0).

- 使用 `cancel` 消息可以取消操作。注意，`cancel` 仅仅是个标记。如果操作已经开始执行，则可能会继续执行下去。
- 可以等待某个操作执行完毕（使用 `waitUntilFinished` 消息）。

• `NSThread`
  - 低级别构造，最大化控制。
  - 应用创建并管理线程。
  - 应用创建并管理线程池。
  - 应用启动线程。
  - 线程可以拥有优先级，操作系统会根据优先级调度它们的执行。
  - 无直接 API 用于等待线程完成。需要使用互斥量（如 `NSLock`）和自定义代码。

> `NSOperationQueue` 是多核安全的。你可以放心地分享队列，从不同的线程中提交任务，而无需担心损坏队列。

4.5 线程安全的代码

贯穿软件开发的职业生涯，我们总是被教导要编写线程安全的代码，这也就是说，如果有多个线程并行地执行同一组指令，不能产生任何副作用。

以下两大类技术可以实现这一点。

- 不要使用可修改的共享状态。
- 如果无法避免使用可修改的共享状态，则确保你的代码是线程安全的。

这些技术说起来容易做起来难。要实现它们有多种选择。

因为应用会包含可修改的共享状态，所以我们需要掌握管理和修改共享状态的最佳实践。

驱动这些最佳实践的一条基本规则是“在代码中保留不变量”。[5]

4.5.1 原子属性

原子属性是实现应用状态线程安全的一个良好开始。如果一个属性是 `atomic`，则修改和读取肯定都是原子的。

这一点很重要，因为这样可以阻止两个线程同时更新一个值，反之则有可能导致错误的状态。正在修改属性的线程必须处理完毕后，其他线程才能开始处理。

所有的属性默认都是原子性的。作为最佳实践，在需要时应该显式地使用 `atomic`。否则使用 `nonatomic` 标记属性。例 4-2 演示了 `atomic` 和 `nonatomic` 属性。

注 5：Stack Overflow,“What Is an Invariant?”(http://stackoverflow.com/a/112088).

例 4-2　atomic 和 nonatomic 属性

```
@property (atomic) NSString *firstName; ❶
@property (nonatomic) NSString *department; ❷
```

❶ 原子属性
❷ 非原子属性

因为原子属性存在开销，所以过度使用它们并不明智。例如，如果能够保证某个属性在任何时刻都不会被多个线程访问，那最好还是将其标记为 `nonatomic`。

使用 `IBOutlet` 时就是一个很好的例子。`@property (nonatomic, readwrite, strong) IBOutlet UILabel *nameLabel` 要比 `@property (atomic, readwrite, strong) IBOutlet UILabel *nameLabel` 更好，因为 UIKit 只允许在主线程中操纵 UI 元素。由于只会在指定的线程内进行访问，除了带来额外开销，将属性设置为 `atomic` 不会带来任何价值。

4.5.2　同步块

即使属性被标记为 `atomic`，最终使用它们的代码仍可能是线程不安全的。原子属性只能阻止并行修改。假设我们有一个 `HPUser` 实体类，可以使用 `HPOperation` 对其进行更新，如例 4-3 所示。

例 4-3　在线程间使用原子属性

```
//一个实体(部分定义)
@interface HPUser

@property (atomic, copy) NSString *firstName;
@property (atomic, copy) NSString *lastName;

@end


//一个服务类(出于简洁的目的省略了声明)
@implementation HPUpdaterService

-(void)updateUser:(HPUser *)user properties:(NSDictionary *)properties {
    NSString *fn = [properties objectForKey:@"firstName"];
    if(fn != nil) {
        user.firstName = fn;
    }
    NSString *ln = [properties objectForKey:@"lastName"];
    if(ln != nil) {
        user.lastName = ln;
    }
}

@end
```

每当用户下拉刷新且数据从服务器返回时，`updateUser:properties:` 方法都会被调用。此外，一个周期性执行的同步任务也会调用该方法。

因此，在某个时间点可能会有多个响应同时尝试更新用户配置文件——可能通过两个 CPU

核心或通过不同的时间片。

思考以下场景：两个响应在不同线程中试图更新用户，名称分别为 Bob Taylor 和 Alice Darji。如果不对属性 `firstName` 和 `lastName` 使用原子更新，则无法确保执行顺序，最终的结果可能是任意组合，其中包括 Alice Taylor 和 Bob Darji。

这个示例只是用于演示，其目的是强调原子属性并不能保证代码一定是线程安全的。

所以我们提供了下一个最佳实践：所有相关的状态都应该在同一个事务中批量更新。

使用 `@synchronized` 指令可以创建一个信号量，并进入临界区，临界区在任何时刻都只能被一个线程执行。例 4-4 展示了改进后的代码。

例 4-4　线程安全的块

```
@implementation HPUpdaterService

-(void)updateUser:(HPUser *)user properties:(NSDictionary *)properties {
    @synchronized(user) { ❶
        NSString *fn = [properties objectForKey:@"firstName"];
        if(fn != nil) {
            user.firstName = fn;
        }
        NSString *ln = [properties objectForKey:@"lastName"];
        if(ln != nil) {
            user.lastName = ln;
        }
    }
}

@end
```

❶ 取得针对 `user` 对象的锁。一切相关的修改都会被一同处理，而不会发生竞争状态。

经过这样的改进，用户最终的名字只能是 Bob Taylor 或 Alice Darji。

> 注意，过度使用 `@synchronized` 指令会拖慢应用的运行速度，因为任何时间都只有一个线程在临界区内执行。

在本例中，我们通过 `user` 对象获取锁。因此，`updateUser:properties` 方法可以从多个线程被调用，不同的用户用不同的线程。当 `user` 对象不同时，该方法仍然能够高并发地执行。结果是代码既实现了高并发，又设置了警戒以防止数据冲突。

> 获取锁的对象是良好定义的临界区的关键。作为经验法则，可以选择状态会被访问和修改的对象作为信号量的引用。

一切到目前为止都还好。但是读取属性时应采取什么策略呢？如果在需要显示 HPUser 对象的全名时，它正在被修改，又该如何处理呢？

4.5.3 锁

锁是进入临界区的基础构件。atomic 属性和 @synchronized 块是为了实现便捷实用的高级别抽象。

以下是三种可用的锁。

- NSLock

  这是一种低级别的锁。一旦获取了锁，执行则进入临界区，且不会允许超过一个线程并行执行。释放锁则标记着临界区的结束。

  例 4-5 展示了如何使用 NSLock。

  例 4-5　使用 NSLock

```
@interface ThreadSafeClass () {
 NSLock *lock; ❶
}
@end

-(instancetype)init {
 if(self = [super init]) {
  self->lock = [NSLock new]; ❷
 }
 return self;
}

-(void)safeMethod {
 [self->lock lock]; ❸

 //线程安全的代码 ❹

 [self->lock unlock]; ❺
}
```

  ❶ 将锁声明为一个私有字段，也可以用属性来表示锁。
  ❷ 初始化锁。
  ❸ 获取锁，进入临界区。
  ❹ 在临界区，任意时刻最多只允许一个线程执行。
  ❺ 释放锁标记着临界区的结束。其他线程现在能够获取锁了。

  NSLock 必须在锁定的线程中进行解锁。

- NSRecursiveLock

  在调用 lock 之前，NSLock 必须先调用 unlock。但正如名字所暗示的那样，NSRecursiveLock 允许在被解锁前锁定多次。如果解锁的次数与锁定的次数相匹配，则认为锁被释放，其他线程可以获取锁。

当类中有多个方法使用同一个锁进行同步，且其中一个方法调用另一个方法时，NSRecursiveLock 非常有用。例 4-6 展示了使用 NSRecursiveLock 的一个例子。

例 4-6　使用 NSRecursiveLock

```
@interface ThreadSafeClass () {
 NSRecursiveLock *lock; ❶
}
@end

-(instancetype)init {
 if(self = [super init]) {
  self->lock = [NSRecursiveLock new];
 }
 return self;
}

-(void)safeMethod1 {
 [self->lock lock]; ❷

 [self safeMethod2]; ❸

 [self->lock unlock]; ❻
}

-(void)safeMethod2 {
 [self->lock lock]; ❹

 //线程安全的代码

 [self->lock unlock]; ❺
}
```

❶ NSRecursiveLock 对象。
❷ safeMethod1 方法获取锁。
❸ 它调用了 safeMethod2 方法。
❹ safeMethod2 从已经获取到的锁再次获取了锁。
❺ safeMethod2 释放了锁。
❻ safeMethod1 释放了锁。因为每个锁定操作都有一个相应的解锁操作与之匹配，所以锁现在被释放，并可以被其他线程所获取。

- NSCondition

  有些情况需要协调线程之间的执行。例如，一个线程可能需要等待其他线程返回结果。NSCondition 可以原子性地释放锁，从而使得其他等待的线程可以获取锁，而初始的线程继续等待。

  一个线程会等待释放锁的条件变量。另一个线程会通知条件变量释放该锁，并唤醒等待中的线程。

  使用 NSCondition 解决标准的生产者—消费者问题。例 4-7 中的代码展示了如何解决这个问题。

例 4-7　使用 NSCondition

```
@implementation Producer

-(instancetype)initWithCondition:(NSCondition *)condition
 collector:(NSMutableArray *)collector { ❶
 if(self = [super init]) {
  self.condition = condition;
  self.collector = collector;
  self.shouldProduce = NO;
  self.item = nil;
 }
 return self;
}

-(void)produce {
 self.shouldProduce = YES;
 while(self.shouldProduce) { ❷
  [self.condition lock]; ❸
  if(self.collector.count > 0) {
   [self.condition wait]; ❹
  }
  [self.collector addObject:[self nextItem]]; ❺
  [self.condition signal]; ❻
  [self.condition unlock]; ❼
 }
}
@end

@implementation Consumer

-(instancetype)initWithCondition:(NSCondition *)condition
 collector:(NSMutableArray *)collector { ❽
 if(self = [super init]) {
  self.condition = condition;
  self.collector = collector;
  self.shouldConsume = NO;
  self.item = nil;
 }
 return self;
}

-(void)consume {
 self.shouldConsume = YES;
 while(self.shouldConsume) { ❾
  [self.condition lock]; ❿
  if(self.collector.count == 0) {
   [self.condition wait]; ⓫
  }
  id item = [self.collector objectAtIndex:0];
  //处理产品
  [self.collector removeItemAtIndex:0]; ⓬
  [self.condition signal]; ⓭
  [self.condition unlock]; ⓮
 }
```

```
}
@end

@implementation Coordinator

-(void)start {
 NSMutableArray *pipeline = [NSMutableArray array];
 NSCondition *condition = [NSCondition new]; ⓯
 Producer *p = [Producer initWithCondition:condition
   collector:pipeline];
 Consumer *c = [Consumer initWithCondition:condition
   collector:pipeline]; ⓰
 [[NSThread initWithTarget:self selector:@SEL(startProducer)
  object:p] start];
 [[NSThread initWithTarget:self selector:@SEL(startCollector)
  object:c] start]; ⓱
  //一旦完成
  p.shouldProduce = NO;
  c.shouldConsume = NO; ⓲
  [condition broadcst]; ⓳
}

@end
```

❶ 生产者的初始化器需要用于协调配合的 NSCondition 对象和用于存放产品的 collector。初始状态设置为不要生产（shouldProduce = NO）。

❷ 生产者会在 shouldProduce 为 YES 时进行生产。其他线程需要将其设置为 NO 以停止生产者的生产。

❸ 获取 condition 的锁，进入临界区。

❹ 如果 collector 中有未消费的产品，则等待，这会阻塞当前线程的执行直到 condition 被通知（signal）为止。

❺ 将生产的 nextItem 送入 collector 以供消费。

❻ 通知其他等待的线程（如果存在）。这里是产品完成生产的标志，并将产品加入到了 collector 中，可供消费。

❼ 释放锁。

❽ 消费者的初始化器需要用于协调配合的 NSCondition 对象和用于存放产品的 collector。在初始化时设置为不消费（shouldConsume = NO）。

❾ 当 shouldConsume 为 YES 时，消费者会进行消费。其他线程可以将其设置为 NO 来停止消费者的消费。

❿ 获取 condition 的锁，进入临界区。

⓫ 如果 collector 中没有产品，则等待。

⓬ 消费 collector 中的下一个产品。确保已经从 collector 中移除它。

⓭ 通知其他等待的线程（如果存在）。这里标识一个产品被消费并从 collector 中移除了。

⓮ 释放锁。

⓯ Coordinator 类为生产者和消费者准备好了输入数据（具体指的是 collector 和 condition）。

⓰ 设置生产者和消费者。

⓱ 在不同的线程中开启生产和消费任务。

⓲ 一旦完成，分别设置生产者和消费者停止生产和消费。

⓳ 因为生产者和消费者线程可能会等待，所以 broadcast 本质上会通知所有等待中的线程。不同的是，signal 方法只会影响一个等待的线程。

4.5.4　将读写锁应用于并发读写

从本节开始，我们将讨论能够实现线程安全的两个方案。本节将介绍避免并发写入的最佳实践，然后在下一节讨论不可变实体。

我们已经学习了 atomic 属性可以用于防护不一致性更新的问题，但有些过于谨慎。如果有多个线程试图读取一个属性，同步的代码在同一时刻只允许单个线程进行访问。因此，使用 atomic 属性会拖慢应用的性能。

这可能是个严重的瓶颈，尤其是当某个状态需要在多个线程间共享，且需要被多个线程访问时。cookie 或登录后的*访问令牌*就是这样的例子。它可以周期性地变化，但会被所有访问服务器的网络请求所调用。

另外一个使用案例是*缓存*。每条缓存的条目可以被应用内的任何地方所访问，并且会因为用户特定的操作而更新。

本质上，我们需要允许并行读取、却与写入互斥的一种机制。这将我们带到了读写锁的话题。它们通常有多个读者、单一写者锁和多个读者、多个写者锁。

读写锁允许并行访问只读操作，而写操作需要互斥访问。这意味着多个线程可以并行地读取数据，但是修改数据时需要一个互斥锁。

GCD 屏障允许在并行分发队列上创建一个同步的点。当遇到屏障时，GCD 会延迟执行提交的代码块，直到队列中所有在屏障之前提交的代码块都执行完毕。随后，通过屏障提交的代码块会单独地执行。我们将这个代码块称为*屏障块*。待其完成后，队列会按照原有行为继续执行。

图 4-2 演示了屏障在多线程环境中执行的效果。块 1 到块 6 可以在多线程间并行执行。但是屏障块单独地执行。唯一的限制是，所有的执行都必须通过并行队列进行执行。

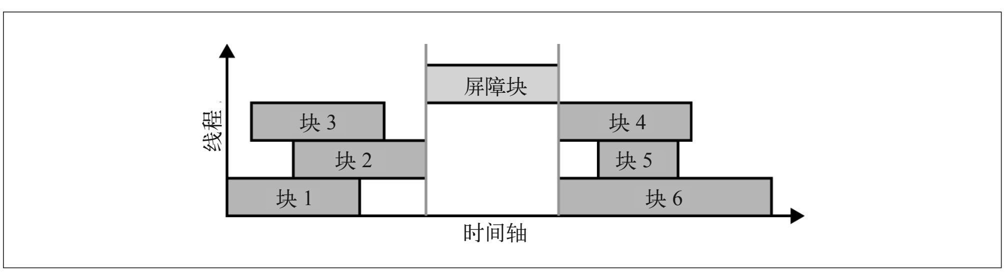


图 4-2：分发块和屏障

要想实现这一行为，我们需要遵循以下步骤。

(1) 创建一个并行队列。

(2) 在这个队列上使用 `dispatch_sync` 执行所有的读操作。
(3) 在相同的队列上使用 `dispatch_barrier_sync` 执行所有的写操作。

你可以使用例 4-8 中的代码来实现高吞吐量且线程安全的模型。

例 4-8　线程安全且高吞吐量的模型

```
//HPCache.h
@interface HPCache

+(HPCache *)sharedInstance;

-(id)objectForKey:(id) key;
-(void)setObject:(id)object forKey:(id)key;

@end

//HPCache.m
@interface HPCache ()

@property (nonatomic, readonly) NSMutableDictionary *cacheObjects;
@property (nonatomic, readonly) dispatch_queue_t queue;

@end

@implementation HPCache

-(instancetype)init {
    if(self = [super init]) {
            _cacheObjects = [NSMutableDictionary dictionary];
            _queue = dispatch_queue_create(kCacheQueueName,
        DISPATCH_QUEUE_CONCURRENT); ❶
    }
    return self;
}

+(HPCache *)sharedInstance {
    static HPCache *instance = nil;

    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
            instance = [[HPCache alloc] init];
    });
    return instance;
}

-(id)objectForKey:(id<NSCopying>)key {
    __block id rv = nil;

    dispatch_sync(self.queue, ^{ ❷
            rv = [self.cacheObjects objectForKey:key];
    });

    return rv;
}
```

```
-(void)setObject:(id)object forKey:(id<NSCopying>)key {
        dispatch_barrier_async(self.queue, ^{ ❸
                [self.cacheObjects setObject:object forKey:key];
        });
}

@end
```

❶ 创建一个自定义的 DISPATCH_QUEUE_CONCURRENT 队列。

❷ 将 dispatch_sync（或 dispatch_async）用于不修改状态的操作。

❸ 将 dispatch_barrier_sync（或 dispatch_barrier_async）用于可能修改状态的操作。

注意，这里的属性被标记为 nonatomic，因为这里有自定义的代码使用了自定义的队列和屏障来管理线程安全。

4.5.5 使用不可变实体

一切看起来都很完美。但是如果需要访问一个正在修改的状态，那将会怎么样呢？

例如，如果缓存被清空，但因为用户执行了一个交互，其中部分状态要求立即被使用，情况将会是怎样的呢？是否存在更有效的机制以管理状态，而不是多个组件试图同时更新状态？

你的团队应该遵循以下的最佳实践。

- 使用不可变实体。
- 通过更新子系统提供支持。
- 允许观察者接收有关数据变化的通知。

这就创建了一个解耦的、可伸缩的系统来管理应用的状态。我们尝试从诸多可能的方案中选取一个予以实现。

首先要清晰地定义模型。在研究案例中，我们定义了以下三个实体。

- HPUser

  表示系统中的一个用户。每个用户有唯一的 ID、拆分为 firstName 和 lastName 的姓名、性别和出生日期。

- HPAlbum

  表示一个相册。每个用户有 0 个或多个相册。每个相册包含唯一的序列号、持有者、名称、创建时间、描述、封面图片链接和点赞（喜欢该相册的用户）。

- HPPhoto

  表示相册内的一张照片。每个相册可以包含 0 张或多张照片。每张照片包含唯一的序列号、所属的相册、用户（上传照片的人）、标题、url 和大小（宽度和高度）。

例 4-9 展示了定义实体的代码。

例 4-9　用于案例研究的实体，表示用户、相册和照片

```
@interface HPUser

@property (nonatomic, copy) NSString *userId;
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
@property (nonatomic, copy) NSString *gender;
@property (nonatomic, copy) NSDate *dateOfBirth;
@property (nonatomic, strong) NSArray *albums;

@end

@class HPPhoto;

@interface HPAlbum

@property (nonatomic, copy) NSString *albumId;
@property (nonatomic, strong) HPUser *owner;
@property (nonatomic, copy) NSString *name;
@property (nonatomic, copy) NSString *description;
@property (nonatomic, copy) NSDate *creationTime;
@property (nonatomic, copy) HPPhoto *coverPhoto;

@end

@interface HPPhoto

@property (nonatomic, copy) NSString *photoId;
@property (nonatomic, strong) HPAlbum *album
@property (nonatomic, strong) HPUser *user;
@property (nonatomic, copy) NSString *caption;
@property (nonatomic, strong) NSURL *url;
@property (nonatomic, copy) CGSize size;

@end
```

有多种方式可以定义填充数据的模型和机制。其中两个常见的方案是：

- 使用自定义的初始化器
- 使用生成器模式

每个方案各有优点。

使用自定义的初始化器意味着会有很长的方法名，从而导致令人讨厌的调用，比如方法 `initWithId:firstName:lastName:gender:birthday:`。况且这只是我们使用模型中部分属性的情况。如果再加五个属性，初始化器会迅速膨胀。

自定义的初始化器还会带来向下兼容的问题。加入更多属性的新版模型无法向下兼容。但是，这也令使用了新版模型的应用能够在编译时知道什么地方发生了改变。

使用生成器模式需要引入外部类进行管理。生成器有 setter 方法，也需要提供与模型数据完全一致的存储。生成器最终也会使用初始化器。

模型的任何更新都需要对生成器及其背后的属性做出相应的改动。

推荐使用生成器模式，因为它支持向下兼容，而且即使不再加入新的属性，生成器也不会破坏应用。新版模型中的新增属性将继续持有其默认值。

第二个方案中的对应代码与例 4-10 类似。这部分代码是基于 Klaas Pieter 使用块（http://www.annema.me/the-builder-pattern-in-objective-c）实现生成器模式的想法而演变的。

例 4-10　使用生成器实现的不可变实体

```
//HPUser.h
@interface HPUserBuilder ❶

@property (nonatomic, copy) NSString *userId;
@property (nonatomic, copy) NSString *firstName;
@property (nonatomic, copy) NSString *lastName;
@property (nonatomic, copy) NSString *gender;
@property (nonatomic, copy) NSDate *dateOfBirth;
@property (nonatomic, strong) NSArray *albums;

-(HPUser *)build;

@end

@interface HPUser ❷

//属性

+(instancetype) userWithBlock:(void (^)(HPUserBuilder *))block;

@end

@interface HPUser () ❸

-(instancetype) initWithBuilder:(HPUserBuilder *)builder;

@end


@implementation HPUserBuilder

-(HPUser *) build { ❹
    return [[HPUser alloc] initWithBuilder:self];
}

@end

@implementation HPUser
-(instancetype) initWithBuilder:(HPUserBuilder *)builder { ❺

    if(self = [super init]) {
        self.userId = builder.userId;
        self.firstName = builder.firstName;
        self.lastName = builder.lastName;
        self.gender = builder.gender;
```

```
        self.dateOfBirth = builder.dateOfBirth;
        self.albums = [NSArray arrayWithArray:albums];
    }
    return self;
}

+(instancetype) userWithBlock:(void (^)(HPUserBuilder *))block { ❻
    HPUserBuilder *builder = [[HPUserBuilder alloc] init];
    block(builder);
    return [builder build];
}

@end


//构建对象,一个例子
-(HPUser *) createUser { ❼
    HPUser *rv = [HPUser userWithBlock:^(HPUserBuilder *builder) {
        builder.userId = @"id001";
        builder.firstName = @"Alice";
        builder.lastName = @"Darji";
        builder.gender = @"F";

        NSCalendar *cal = [NSCalendar currentCalendar];
        NSDateComponents *components = [[NSDateComponents alloc] init];
        [components setYear:1980];
        [components setMonth:1];
        [components setDay:1];
        builder.dateOfBirth = [cal dateFromComponents:components];

        builder.albums = [NSArray array];
    }];

    return rv;
}
```

❶ 生成器。
❷ 模型提供了类方法 `userWithBlock:`。例 4-9 包含了声明的全部属性。
❸ 模型的私有扩展——自定义的初始化器。
❹ `build` 方法的实现。
❺ 模型自定义初始化器的实现。
❻ `userWithBlock:` 方法的实现。
❼ 用生成器创建对象的使用示例。

注意，前面的代码具有以下优点。

- 模型总是向下兼容。新版的模型生成器包含了新增属性,但不会破坏 `createUser` 的代码。
- 生成器可以被直接创建。模型的消费者可以初始化生成器，并调用 `build` 方法创建模型对象。
- 生成器的创建和处理可以留给内部核心完成。模型的消费者可以使用类方法 `userWithBlock:` 而无需初始化或亲自调用 `build` 方法。

4.5.6 使用集中的状态更新服务

下一步我们需要一个更新服务，以更新客户端状态。更新服务可能需要连接服务器，在执行本地更新前进行验证，如加入或更新一条记录、确认好友的申请或上传一张照片。从 UI 的视角来看，在短暂的期间内，你需要向用户展示一个进度条或其他的指示器，以通知用户有关状态的实际情况。

在案例中，我们将 `HPUserService`、`HPAlbumService`、`HPPhotoService` 分别用于服务 `HPUser`、`HPAlbum`、`HPPhoto` 对象。

更新状态很棘手，因为状态是不可变的。有些矛盾，不是吗？解决办法是让状态的生成器接收一个后续可修改的输入状态。

为了对 `HPUser` 实现这一点，我们可以在 `HPUserBuilder` 上创建一个辅助初始化器，以便接收输入的对象。

例 4-11 中的代码展示了改进后的 `HPUserBuilder` 类，以便支持对之前创建的 `HPUser` 对象进行修改，同时还创建了 `HPUserService` 类来获取和更新对象。类似的基础改造也适用于 `HPAlbum` 和 `HPPhoto` 实体。这段代码演示了用于用户和相册实体的服务，通常用于以下两个场景：

- 从服务器获取数据并导致本地状态的更新；
- 更新本地和远程的状态，例如，通过用户的交互。

例 4-11 用于用户和相册对象的服务

```
//HPUserBuilder.h
@interface HPUserBuilder

-(instancetype) initWithUser:(HPUser *)user;

@end

@interface HPUserBuilder

-(instancetype) initWithUser:(HPUser *)user { ❶
    if(self = [super init]) {
        self.userId = builder.userId;
        self.firstName = user.firstName;
        self.lastName = user.lastName;
        self.gender = user.gender;
        self.dateOfBirth = user.dateOfBirth;
        self.albums = user.albums;
    }
    return self;
}

@end

//HPUserService.h
@interface HPUserService
```

```
+(instancetype)sharedInstance; ❷
-(void)userWithId:(NSString *)id completion:(void (^)(HPUser *))completion;
-(void)updateUser:(HPUser *)user completion:(void (^)(HPUser *))completion;

@end

//HPUserService.m
@interface HPUserService

@property (nonatomic, strong) NSMutableDictionary *userCache; ❸

@end

@implementation HPUserService

-(instancetype) init { ❹
    if(self = [super init]) {
        self.userCache = [NSMutableDictionary dictionary];
    }
    return self;
}

-(void)userWithId:(NSString *)id completion:(void (^)(HPUser *))completion { ❺
    //检查本地缓存或从服务器提取
    HPUser *user = (HPUser *)[self.userCache objectForKey:id];
    if(user) {
        completion(user);
    }

    [[HPSyncService sharedInstance] fetchType:@"user"
        withId:id completion:^(NSDictionary *data) { ❻
        //使用HPUserBuilder,分析数据并构建
        HPUser *userFromServer = [builder build];
        [self.userCache setObject:userFromServer forKey:userFromServer.userId];
        callback(userFromServer);
    }];
}

-(void)updateUser:(HPUser *)user completion:(void (^)(HPUser *))completion { ❼
    //可能会要求更新到服务器
    [[HPSyncService sharedInstance] updateType:@"user"
        //使用HPUserBuilder,分析数据并构建
        HPUser *updatedUser = [builder build];

        [self.userCache setObject:updatedUser forKey:updatedUser.userId]; ❽
        [HPAlbumService updateAlbums:updatedUser.albums]; ❾
        completion(updatedUser);
    }];
}

@end
```

❶ HPUserBuilder 现在有了另一个自定义初始化器。它以 HPUser 对象作为输入参数，并利用 user 对象的值对自身进行初始化。可以通过属性的 setter 方法修改状态，并最终使用

build 方法构造新的对象。注意，虽然状态被修改，但旧的对象却没有发生任何改动。这也意味着，如果旧的对象正被其他实体（如视图控制器）所使用，则需要进行替换。我们将在下一节探讨状态变更的通知。

❷ HPUserService 遵循了单例模式，并可以使用 sharedInstance 方法。出于简洁的目的省略了代码，但我们知道该如何实现良好、安全的单例。通常而言，在实体或服务层次使用单例并非明智的选择，因为这会带来紧密的耦合并影响 mock 框架的使用。此时，使用可配置的工厂要优于使用单例。工厂可以创建可销毁的单例。我们将在第 10 章回顾这一话题。

❸ 作为快速原型，服务还持有了自己所创建的用户对象的缓存。但是，将状态和缓存逻辑混在一起并不是个好主意。应当总是保持状态与任何其他*聪明的*代码分离。你可能想让模型*越蠢越好*。

❹ HPUserService 初始化器被复写来初始化缓存。这只是个权宜之计，因为我们的目的是集中讨论不变对象如何才能比可变对象更好地服务应用，而后者的状态可在应用的不同地方修改。在实际的应用中，服务对象会访问状态，后者可以作为任何处理或更新的输入，此外，服务对象还会访问网络操作以保持与服务器的同步。

❺ 使用 userWithId:completion: 方法可以获取有给定 id 的用户。如果该对象存在于本地状态，则会被返回。否则，它会请求服务器获取详细信息。一旦完成，completion 回调会被触发，以通知调用者对象可用。

❻ 假设这里存在一个同步服务 HPSyncService。该服务会从服务器获取数据。还假设服务器发送了 JSON 对象[6],并被反序列化为 NSDictionary 对象。抽取属性和填充生成器的代码被忽略了。一旦数据可用，我们还将更新本地的缓存，以避免后续的服务器请求。

❼ 使用 updateUser:completion: 方法可以更新用户的状态。

❽ 更新本地状态可能需要同步修改服务器。

❾ 一旦通知了服务器，本地缓存就要被更新。因为用户对象持有相册，所以相册服务也用于更新相关的相册。具体来讲，关联的持有者对象要指向更新后的用户对象。旧的用户对象需要被析构。注意，这里提供的解决方案伸缩性较差：如果其他实体对象需要更新自身，那该怎么办呢？接下来我们将会解决这个问题。

实体之间的交叉引用是一个需要注意的点。用户有一个相册列表，每个相册都有一个主人。类似地，相册中有照片列表，而每个照片都有它所隶属的相册。我们甚至还没有为照片的评论建模，评论会包含写评论的时间、内容和写评论的作者。

不管它们是强还是弱，创建包含此类交叉引用的不可变对象被刻意忽略了。我们需要用户对象早于相册创建完成，反之亦然。这是个矛盾的局面。

解决之道是，如果没有具体标记为不可变，则保持对象可变。这就是所谓的*冰棒不变性*。[7] 要想实现这一点，你需要一个特殊的方法 freeze 或 markImmutable。为了能够使用这一结构，你需要自定义的 setter 方法，以便在允许修改前检查对象是否为不可变。

现在我们可以解决这个死锁。首先，让 HPAlbum 在设置持有者之前允许被修改。我们创建

注 6：你可能还想考虑其他格式，如 Protobuf、Thrift 或 Avro。

注 7：Stack Overflow：“How to Design an Immutable Object with Complex Initialization”（http://bit.ly/1FuVRGI）。

HPUser 对象，并设置 HPAlbum 对象的持有者。随后，我们调用 HPAlbum 对象的 freeze 方法。当全部相册创建完成之后，我们将其设置为 HPUser 对象的 albums 属性。最终，我们调用 HPUser 对象的 freeze 方法。

代码效果如例 4-12 所示。HPUser 被更新为拥有读 / 写属性，并在标记为不可变之前是可变的。而且可以设想的是，绝大部分的使用场景都不再需要生成器了，因为属性本身是读 / 写属性。

例 4-12　冰棒不可变的实体

```
//HPUser.h
@interface HPUser

@property (nonatomic, copy) NSString *userId; ❶
@property (nonatomic, copy) NSString *firstName;
-(void) freeze; ❷

@end

//HPUser.m
@interface HPUser ()

@property (nonatomic, copy) BOOL frozen; ❸

@end

@implementation HPUser

@synthesize userId = _userId; ❹
@synthesize firstName = _firstName;

-(void) freeze { ❺
    self.frozen = YES;
}

-(void) setUserId:(NSString *)userId { ❻
    if(!self.frozen) {
        self-&gt;_userId = userId;
    }
}

-(void) setFirstName:(NSString *)firstName {
    if(!self.frozen) {
        self-&gt;_firstName = firstName;
    }
}

//……省略了其他的setter

@end

//创建对象
-(HPUser *)sampleUser { ❼
    HPUser *user = [[HPUser alloc] init];
```

```
    user.userId = @"user-1";
    user.firstName = @"Bob";
    user.lastName = @"Taylor";
    user.gender = @"M";

    HPAlbum *album1 = [[HPAlbum alloc] init];
    album1.owner = user; ❽
    album1.name = @"Album 1";
    //……其他属性
    [album1 freeze]; ❾

    HPAlbum *album2 = [[HPAlbum alloc] init];
    album2.owner = user;
    album2.name = @"Album 2";
    //……其他属性
    [album2 freeze]; ❿

    user.albums = [NSArray arrayWithObjects:album1, album2, nil];
    [user freeze]; ⓫

    return user;
}
```

❶ 属性不再是 `readonly`，而是 `readwrite`（隐式）。

❷ 我们加入了 `freeze` 方法，将对象标记为不可变。对象默认是可变的。

❸ 一个用于跟踪对象的不可变状态的标签。

❹ 因为将编写自定义的 setter 方法，所有我们需要 `@synthesize` 并告诉编译器使用隐藏的 iVar。

❺ `freeze` 方法的实现标记对象为不可变。

❻ 自定义 setter。首先检查对象是否为可变。如果是，则更新。如果不是，则不更新。你还可以在开发阶段抛出一个异常，来保障合法的调用和识别任何错误的代码。

❼ 演示新 API 使用的示例代码。

❽ 将一个用户设置为相册的持有者。此时，两个对象都是可变类型的。

❾ 将 `HPAlbum` 对象标记为不可变。

❿ 将 `HPUser` 对象标记为不可变。注意，只有在这之前才可以使用不可变的相册对象。

虽然对象在短时间内可以被修改，但我们能够确保可变性是短暂的，且仅限于创建对象的线程。在将对象从创建方法送入需要它们的应用状态之前，你必须确保它们已经被标记为不可变。

4.5.7 状态观察者与通知

上一节留给我们一个未解决的问题：如果对象被更新，那么我们应该如何同步更新依赖？或者换个角度，追踪状态变更的最佳实践是什么？

要跟踪变更，你有如下选择：

- 键 – 值观察
- 通知中心
- 自定义的方案

我们在第 2 章中介绍过前两个方案。键 – 值观察非常适合用来跟踪对象的属性。但是在我们的方案中，因为对象是不可变，而且我们要替换整个对象，所以键 – 值观察没有用武之地。因此，观察者不会收到任何的回调通知。

通知中心是优秀的解决方案。它提供了实用的功能，可以满足大多数的情况。但问题在于它最终会放大应用的复杂性。例如，按相册标识号码过滤更新通知或尽可能直接地将变化抛向 UI。

这就需要自定义的解决方案出场了。要实现这个目标，我们将切换到响应式编程的风格。

响应式编程是基于异步数据流的编程方式。[8] 流是廉价且无处不在的，一切都可以是流：变量、用户输入、属性、缓存、数据结构，等等。

ReactiveCocoa 库（https://github.com/ReactiveCocoa/ReactiveCocoa）实现了在 Objective-C 中进行响应式编程。它不仅可以实现对任意状态的观察，还提供了高级的分类扩展，以便同步更新 UI 元素（如 `UILabel`）或响应视图的交互（如 `UIButton`）。

> **函数响应式编程与 ReactiveCocoa**
>
> 应用通常会消费、制造和更新数据。响应式编程是一种编程范式，能够表示数据流，而无需担心副作用或对其他并发执行的任务造成影响。
>
> 响应式编程背后的核心思想是随时间流逝体现数据的值。使用动态值的数据流会导致这些值随着时间而变化。
>
> 函数响应式编程（functional reactive programming，FRP）允许响应式编程使用内建的块方法进行函数式编程，如 map、reduce、filter、merge，等等。
>
> ReactiveCocoa 的灵感来自 FRP。因为多组件的应用以高内聚的方式工作，所以组件状态的使用和更新就有了很强的关联。正因为这一点，创建一个低耦合、高内聚的响应式系统才显得尤为重要。

我们将使用 ReactiveCocoa 通知观察者模型发生了变化。观察者可以创建在任何地方。

为了说明目的，我们将在每个用户对象的创建和更新操作期间添加通知。我们也会为相册服务添加观察者，以监视相册主人（用户）的变化。出于完整性的目的，在 UI 中为用户监控相册列表的变化。例 4-13 展示了代码的相关部分。

例 4-13　观察者和通知

```
//HPUserService.m
-(RACSignal *)signalForUserWithId:(NSString *)id { ❶
    @weakify(self);
    return [RACSignal
        createSignal:^RACDisposable *(id<RACSubscriber> subscriber) { ❷
        @strongify(self);
```

注 8：The introduction to Reactive Programming you’ve been missing (https://gist.github.com/staltz/868e7e9bc2a7b8c1f754#reactive-programming-is-programming-with-asynchronous-data-streams).

```
        HPUser *userFromCache = [self.userCache objectForKey:id];
        if(userFromCache) {
            [subscriber sendNext:userFromCache];
            [subscriber sendCompleted];
        } else {
            //假设HPSyncService也遵循FRP风格
            [[[HPSyncService sharedInstance]
                loadType:@"user" withId:id]
                subscribeNext:^(HPUser *userFromServer) {
                    //也更新本地缓存和通知
                    [subscriber sendNext:userFromServer];
                    [subscriber sendCompleted];
                } error: ^(NSError *error) {
                    [subscriber sendError:error];
                }];
        }

        return nil;
    }]
}

-(RACSignal *)signalForUpdateUser:(HPUser *)user { ❸
    @weakify(self);
    return [RACSignal
        createSignal:^RACDisposable *(id<RACSubscriber> subscriber) { ❹
        //更新服务器
        [[[HPSyncService sharedInstance]
            updateType:@"user" withId:user.userId value:user]
            subscribeNext:^(NSDictionary *data) {
                //使用HPUserBuilder,分析数据并构建
                HPUser *updatedUser = [builder build];

                @strongify(self);
                var oldUser = [self.userCache objectForKey:updatedUser.userId];
                [self.userCache setObject:updatedUser forKey:updatedUser.userId];
                [subscriber sendNext:updatedUser];
                [subscriber sendCompleted];
                [self notifyCacheUpdatedWithUser:updatedUser old:oldUser]; ❺
            } error: ^(NSError *error) {
                [subscriber sendError:error];
            }];
    }];
}

-(void)notifyCacheUpdatedWithUser:(HPUser *)user old:(HPUser *)oldUser { ❻
    NSDictionary *tuple = {
        @"old": oldUser,
        @"new": user
    };
    [NSNotificationCenter.defaultCenter
        postNotificationName:@"userUpdated" object:tuple]; ❼
}

-(RACSignal *)signalForUserUpdates:(id)object { ❽
    return [[NSNotificationCenter.defaultCenter
```

```
        rac_addObserverForName:@"userUpdated" object:object] ❾
        flattenMap:^(NSNotification *note) {
            return note.object;
        }];
}

//在应用的其他地方
-(void)retrieveAUser:(NSString *)userId { ❿
    [[[HPUserService sharedInstance]
        signalForUserWithId:userId]
        subscribeNext:^(HPUser *user) { ⓫
            //处理用户,或者更新
        } error:^(NSError *) {
            //向用户显示错误
        }];
}

-(void)updateAUser:(HPUser *)user { ⓬
    [[[HPUserService sharedInstance]
        signalForUpdateUser:user]
        subscribeNext:^(HPUser *user) { ⓭
            //处理用户,或者更新
        } error:^(NSError *) {
            //向用户显示错误
        }];
}

//监听用户更新
-watchForUserUpdates { ⓮
    [[[HPUserService sharedInstance]
        signalForUserUpdates:self] ⓯
        subcribeNext:^(NSDictionary *tuple) { ⓰
            //用值做一些事情
            HPUser *oldUser objectForKey:@"old";
            HPUser *newUser objectForKey:@"new";
    }];
}
```

❶ `signalForUserWithId` 方法并未将块作为参数，而是返回了一个可以被链式调用的 Promise。这里还使用了 2.15 节中介绍的 `@weakify` 和 `@strongify` 两个宏。

❷ 信号的代码与 `userWithId` 中原来的代码格外相似，不同之处是这里使用了 `RACSubscriber` 和一个信号。

假设 `HPSyncService` 类中的 `loadType:withId` 方法也返回一个信号，即一个 `RACSignal`。

❸ `signalForUpdateUser` 方法更新一个 `HPUser` 对象。

❹ 这里创建了 `RACSignal`。

❺ 当用户被更新，你不仅需要通知直接订阅者，还要通知观察者更新缓存。

❻ `notifyCacheUpdatedWithUser:old:` 广播了用户对象的变化。

❼ 这里使用 `NSNotificationCenter` 是为了简化。这个方法不宜暴露给 `HPUserService` 类的用户。这是个扩展方法。

❽（在 HPUserService.h 文件中的）公开的方法是 `signalForUserUpdates:`。

❾ 这里使用了 ReactiveCocoa 框架提供的分类扩展 `rac_addObserverForName`，以订阅 `userUpdated` 通知。它也会实际从 `NSNotification` 中抽取 `NSDictionary`，后者由旧的和新的用户对象所组成。
❿ `retrieveAUser:` 方法演示了获取用户的示例代码。
⓫ `subscribeNext:` 块是接收 `user` 对象的地方。
⓬ `updateAUser:` 方法演示了更新用户的示例代码。
⓭ `subscribeNext:` 块是接收 `user` 对象的地方。
⓮ `watchForUserUpdates:` 方法展示了在用户缓存中观察变化的示例代码。
⓯ 它使用 `signalForUserUpdates:` 方法监听用户缓存发生改变的通知。
⓰ `subscribeNext:` 块会传入包含新旧用户对象的 `NSDictionary`。

这里的优点在于，如果未来 `signalForUserUpdates:` 的实现不再依赖 `NSNotificationCenter`，那将不会影响 `watchForUserUpdates:` 方法的实现方案。

使用这个库的主要动机是，它可以帮助我们实现一个观察变化的系统，这个系统有着低耦合、高伸缩性、自包含且适用于通用目的的特点。更重要的是，它为链接（使用 `RACSignal`）提供了 Promise，使得我们能够写出更易理解和维护的代码。它还提供了更加简便的方案以实现与 UI 元素的交互，有些元素会在后续章节中体现出来。简而言之，它提供了许多样本代码，没有它我们就得自己编写这些代码，而且要做得更多。

使 Facebook 的新闻流在 iOS 上加速 50%

2012 年，Facebook 将其新闻流从 HTML5 迁移到了原生的 iOS 应用以优化性能。但是久而久之，随着包括小组、主页和时间轴的其他部分都迁移到了原生地，新闻流拉低了整体的性能。分析诊断显示根本原因在数据层。[9]

因此，模型层基于三个原则进行了重写：

- 不可变性
- 非规范化存储
- 异步，选择性一致性

这也意味着放弃了 Core Data 框架，该框架能保证很强的数据一致性，但却会导致性能损失。[10]

4.5.8 异步优于同步

在上一节中，我们了解了应该优先选用 Promise。本节提供了一些针对异步代码的更深入讨论。

一个有说服力且令人印象深刻的理由支持我们首选异步方式而不是同步方式。且这与同步

注 9：Facebook,“Making News Feed Nearly 50% Faster on iOS”(http://bit.ly/faster--ios).
注 10：“Facebook’s iOS Architecture - @Scale 2014 - Mobile”().

有关。我们在 4.5 节中讨论了使用屏障，并学习了如何将 dispatch_sync 用于并发读取。

接下来我们简单分析一下例 4-14 中的代码。

例 4-14　在实际场景中使用 dispatch_sync

```
//场景A
dispatch_sync(queue, ^() {
    dispatch_sync(queue, ^() {
        NSLog(@"nested sync call");
    });
});

//场景B
-(void) methodA1 {
    dispatch_sync(queue1, ^() {
        [objB methodB];
    });
}

-(void)methodA2 {
    dispatch_sync(queue1, ^() {
        NSLog(@"indirect nested dispatch_sync");
    });
}

-(void) methodB {
    [objA methodA2];
}
```

在例 4-14，场景 A 演示了一个假设的场景，在这个场景中使用分发队列调用了一个嵌套的 dispatch_sync。这会导致死锁。嵌套的的 dispatch_sync 不能分发到队列中，因为当前线程已经在队列中且不会释放锁。

场景 B 演示了更相似的场景。类 A 有两个使用了相同队列的方法（methodA1 和 methodA2）。前一个方法对某个对象调用了 methodB 方法，后续会反调回来。最终结果还是死锁。一个原本很有用的方法 dispatch_get_current_queue 已经被弃用很久了。[11]

方案之一是使用 dispatch_queue_set_specific 和 dispatch_queue_get_specific 方法（http://bit.ly/1NOj8fo），但这仍然会让代码变得糟糕。

要想实现线程安全、不死锁且易于维护的代码，强烈建议使用异步风格。使用 Promise 是最好的方式。ReactiveCocoa（参见 4.5 节）为 Objective-C 引入了 FRP 风格。dispatch_async 不受这一行为的影响。

注 11：dispatch_get_current_queue, Developer Tools Manual Page (http://apple.co/1RPh8SH).

PromiseKit

PromiseKit（https://github.com/mxcl/PromiseKit）是支持使用 Promise 的另外一个库。它甚至做得更好，因为它可以帮助避免代码向右漂移。

例 4-15　向右漂移的 Promise

```
[[[[
    [[HPNetworkService sharedInstance] promise:rq1]
        subcribeNext:^(id data1) {
            return [[HPNetworkService sharedInstance] promise:rq2];
        }]
    subscribeNext:^(id data2) {
        return [[HPNetworkService sharedInstance] promise:rq3];
    }]
    subscribeNext:^(id data3) {
        // 此处有三级缩进
        // 查看开括号 '[[[['
    }]
}];
```

例 4-16　不向右漂移的 Promise

```
[NSURLConnection promise:rq1].then(^(id data1){
    return [NSURLConnection promise:rq2];
}).then(^(id data2){
    return [NSURLConnection promise:rq3];
}).then(^(id data3){
    // 哇！代码看起来很连贯！
});
```

注意，在例 4-15 中，如果多个 Promise 被链式调用，则会有一系列的中括号（[），如果你是一位每个中括号一行且增加一个缩进的程序员，则代码会立刻向右漂移许多。另一方面，在例 4-16 中，代码总是在第一列对齐。

PromiseKit 也提供了优雅的错误处理能力。强烈建议深入研究 PromiseKit。

4.6　小结

无法想象哪个应用可以不使用并行编程，即使像动画一样简单的操作也需要多任务。一切耗时的任务（如网络和 I/O）都必须在后台线程内完成。

通过对现有不同方案（即线程、GCD、操作和队列）的深入分析，你现在应该能够选择一个更加符合具体场景的方案进行工作。

选择正确的方案来保证代码的线程安全，是实现应用状态正确的关键。使用信号量同步访问代码块非常重要；使用读 – 写锁实现高吞吐量的读和有保护的写同样重要。

通过阅读这一部分，你已经熟悉了一些核心优化技术，如内存管理、电量使用和并行编程。现在你可以对应用的模型和业务逻辑层进行优化了。

第三部分

iOS性能

第二部分为创建高性能应用奠定了基础。高性能应用就是了解资源利用并遵循优化的最佳实践的应用。虽然我们的讨论围绕着 iOS 应用开发，但通用原则适用于任何 Objective-C 应用。

这一部分的章节将会关注 iOS 应用开发中特有的选项和技术。我们将主要探讨以下主题：

- 应用的生命周期
- UI
- 网络
- 数据共享
- 安全

第 5 章

应用的生命周期

iOS 应用启动时会调用 UIApplicationMain 方法，并传入 UIApplicationDelegate 类的引用。委托接收应用范围的事件，并且有明确的生命周期，application:didFinishLaunchingWithOptions: 方法表明应用已经启动。关键组件的初始化就发生在这个方法中，如崩溃上报、网络、日志以及埋点的初始化。此外，初次启动或恢复前置状态以便后续启动时，还可能会执行一些一次性的初始化操作。

应用的窗口有一个 rootViewController，可以驱动展示给用户的 UI。对应的 UIViewController 对象同样具有明确的生命周期。

应用启动过程中的一系列活动会影响初始加载时间，因此，为了获得更好的用户体验，必须使初始化活动的数量最小化。但并不能任性地移除初始化的任务，因为如果因此而导致应用启动之后的后续操作变得更慢，肯定会惹恼用户。

本章将深入讨论应用的生命周期。我们会将应用开发人员使用事件回调的目的，与事件回调的主要意图及对应用性能的影响进行对比。我们也将回顾一些可以取悦用户的技术、秘诀和窍门。

我们将在第 6 章中探索 UIViewController 的生命周期。

5.1　应用委托

应用委托通常是应用创建的第一个对象。它为应用提供一些环境变量，其中包括应用启动的详细信息、远程通知、深层链接，等等。

如果需要回顾应用的结构和执行状态，可以参阅 http://apple.co/1IV94sL。

图 5-1 展示了在不同状态之间切换时对应的应用委托回调。

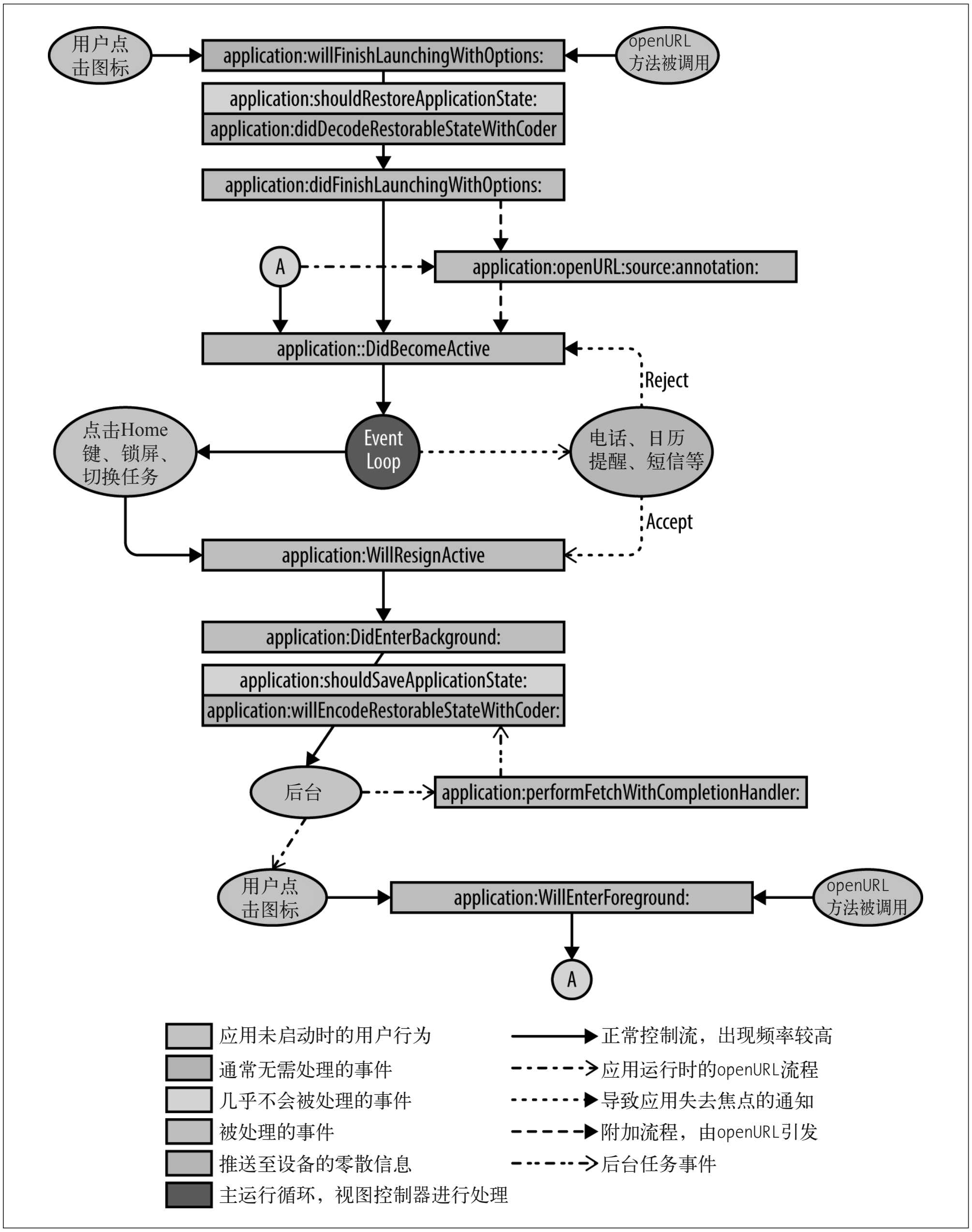


图 5-1：应用委托回调的调用

虽然图 5-1 整体看起来比较复杂，但每个部分都是我们熟悉的内容，`application:didDecodeRestorableStateWithCoder:` 和 `application:willEncodeRestorableStateWithCoder:` 这两个方法可能除外，因为它们很少被调用。但是应用通常都会创建本地状态管理，可以从那里

恢复之前的状态。将这些方法都加进来是为了让整个图看起来更加完整。

还有一些其他的事件回调并未在图中列举出来，与推送通知有关的回调被省略了，我们将在 5.3 节中探讨。

我们将一次分析一个回调，检查回调中的代码，并寻找是否存在更好的实现方式。

5.2 应用启动

著名的 `application:didFinishLaunchingWithOptions:` 方法是应用启动时最核心的地方。此处不能发生任何错误，且绝不能发生崩溃，否则应用将无法正常使用，直到下次升级。一旦发生这种情况，如果应用对用户来说并非至关重要，那用户肯定就会放弃使用这款应用。

上述方法会载入所有的依赖，并初始化应用的核心。在启动时，你必须使该方法的执行时间尽可能短，因为你不希望用户等待一段时间 UI 才展现出来。同时你也不希望应用因此被贴上“笨拙”“庞大”“缓慢”这样的标签，当然也不想在 App Store 里得到糟糕的评价。

应用有四种启动类型。

- 首次启动
  安装应用后的首次启动。此时没有之前的状态，也没有本地缓存。

  这意味着将会出现以下两种情况中的一种：没有需要加载的内容（因此加载时间会缩短），或者需要从服务器上下载初始数据（可能需要很长的加载时间）。

  在应用首次启动时，你可以选择提供引导图来总结应用的功能和用法。图 5-2 展示了 Dropbox 的引导图。

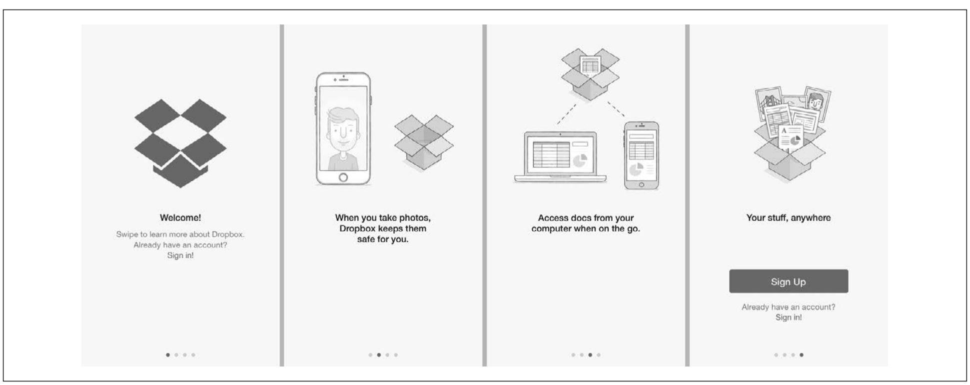


图 5-2：Dropbox 的引导图

- 冷启动
  应用后续的启动。在启动期间，可能需要恢复原来的状态，例如，游戏中达到的最高等级、消息应用中的聊天记录、新闻应用中上一次同步的文章、已登录用户的证书，或者仅仅是用户已经使用过的引导图标记符。

  图 5-3 展示了冷启动时的 Facebook 应用。注意一下它是如何在从服务器获取更新的同时快速载入缓存内容的。

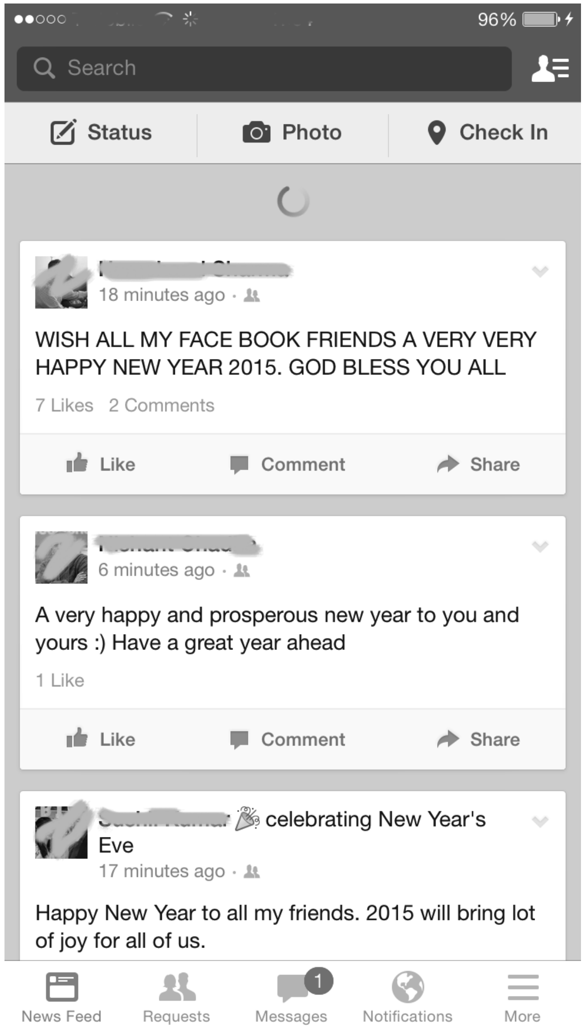


图 5-3：冷启动时的 Facebook 应用

- 热（重）启动

  这是指当应用处于后台，但并未被挂起或关闭时，用户切换至应用而触发的启动。在这种情况下，当用户通过点击应用图标或深层链接（参见 8.1 节）返回应用时，不会触发启动时的回调，而是直接用 `applicationDidBecomeActive:`（或 `application:openURL:source:annotation:`）回调。

  通常来说，这种情况和继续执行没什么区别，只是视图控制器可能需要处理一些额外的事件，对此，我们将在本章后续部分进一步探讨。

- 升级后的启动
  应用升级以后的启动。通常而言，升级后的启动与冷启动没有差别。但是，不同的启动叫法表明了本地存储发生变化的时刻是不同的，这些变化包括模式、内容、之前版本挂起的同步操作，以及内部的 API/ 默认依赖。

5.2.1 首次启动

首次启动时，应用通常会执行多个任务：

- 加载应用的默认项（NSUserDefaults、捆绑的配置等）
- 检查私有 / 测试版本
- 初始化应用标识符，包括但不限于对匿名用户使用的供应商标识符（Identifier for Vendor，IDFV）、广告标识符（Identifier for Advertiser，IDFA）等
- 初始化崩溃报告系统
- 建立 A/B 测试
- 建立分析方法
- 使用操作或 GCD 建立网络
- 建立 UI 基础设施（导航、主题、初始 UI）
- 显示登录提示或从服务器加载最新内容及其他更新
- 建立内存缓存（如图片缓存）

上述列举的内容只是应用在首次启动时可能执行的任务。其中一些还会在后续启动中执行。问题是，任务数量的快速增加必然会导致应用的启动速度变慢。

如果以某一应用为背景介绍这些任务，类似的代码如例 5-1 所示。

例 5-1　应用的启动代码

```
-(BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions { ❶

    NSString *deviceId = [[[UIDevice currentDevice]
            identifierForVendor] UUIDString]; ❷

    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    BOOL firstLaunch = ![defaults boolForKey:@"appLaunched"]; ❸
    if(firstLaunch) {
        [defaults setBool:YES forKey:@"appLaunched"]; ❹
        [defaults synchronize];

        //将设备注册到服务器 ❺
    }

    //用设备ID建立A/B测试 ❻

    //……更多建立 ❼
    [Flurry startSession:@"API_KEY"];
    [[NSURLCache sharedURLCache] setMemoryCapacity:(8 * 1024 * 1024)];
    [[NSURLCache sharedURLCache] setDiskCapacity:(50 * 1024 * 1024)];
    [SDImageCache sharedImageCache].maxCacheSize = 8 * 1024 * 1024;
```

```
    NSString *accessToken = nil; ❽
    if(!firstLaunch) {
        accessToken = [defaults stringForKey:@"accessToken"];
    }
    if(accessToken) { ❾
        //用户登录
    } else {
        if(firstLaunch) { ❿
            //首次启动
        } else { ⓫
            //用户未登录
        }
    }

    return YES;
}

-(BOOL)applicationDidBecomeActive:(UIApplication *)application { ⓬

#ifdef RELEASE_BETA ⓭
    [[BITHockeyManager sharedHockeyManager]
        configureWithIdentifier:@"API_KEY"];
    [[BITHockeyManager sharedHockeyManager] startManager]; ⓮
#endif
}
```

❶ 每个应用启动时都会调用一次 `application:didFinishLaunchingWithOptions:` 回调。
❷ 获取 IDFV 来唯一地跟踪此设备。
❸ 确定应用是初次启动还是之前就已经启动过了。
❹ 设置标记，表明应用之前已经启动过了。
❺ 可以将 ID 发送到服务器进行匿名登记，或用来跟踪唯一用户或设备数量。
❻ 一些子系统可能需要 ID，例如，为了进行 A/B 测试。
❼ 其他设置：分析方法、网络缓存、图片缓存，等等。
❽ 追踪用户登录状态的访问令牌。
❾ 如果访问令牌存在，则表明用户已经登录过。如果用户修改了密码或从其他设备远程登出，则需要刷新令牌。为简洁起见，此处省略了部分代码。
❿ 如果访问令牌不存在，则存在两种情况：其一，应用是初次启动，此时或许需要展示应用引导图；其二，用户在以前的应用会话期间并未登录。
⓫ 如果用户没有登录且应用也不是初次启动，你可以直接展示登录表单。
⓬ 一些配置项可能要等应用回到前台时（激活状态）才能被设置。
需要注意的是，每次应用从后台切换至前台时，此方法都会被调用，并非只是启动时。因此，你要确保不运行动画以免引入重复延迟，因为这可能会惹恼用户。
⓭ `RELEASE_BETA` 不是标准的标记，而是自定义标记，用来区分是 App Store 版本还是私有发布。为了保证正常运行，你需要创建多个配置项 / 目标，如图 5-4 所示。
⓮ 在本例中，HockeyKit（https://www.hockeyapp.net/）设置已经完成，应用可以执行其他任务了。



图 5-4：多种配置

注意，从各个方面来看，这个列表都还不够完整。广告相关的初始化、日志、应用安装的属性、单点登录等其他任务都没有展示出来。其实这很大程度上取决于应用的需求和结构。

> 虽然每个子系统都有较高的性能，但一起使用时，整体的性能可能会下降。例如，如果多个组件试图同时从文件系统读取，则一定会导致整体变得迟缓。

这样就会出现之前提到的情景——子系统的初始化需要大量的时间，而它们彼此之间可能又存在依赖。例如，主题可能依赖 A/B 测试，数据同步可能依赖令牌是否有效（又名登录）。

如果这些（可能是强制的）初始化加起来要耗费很长时间，我们该如何优化呢？

这个问题没有明确的答案。其中一个方法是退一步，先确定要展示应用 UI 所需的最低要求，再从此处着手。

你可以遵循下述具体步骤，拆解任务列表，从而获得更高的性能。

(1) 确定在展示 UI 前必须执行的任务。

如果应用是第一次启动，那么没有必要加载任何用户偏好，如主题、刷新间隔、缓存大小等。此时是没有任何自定义值的。初始缓存肆意增长也是没问题的，因为它的增长不会超过最终的限制值。

崩溃报告系统应该第一个被初始化。

(2) 按顺序执行任务。

排序是非常重要的，因为任务之间可能具有相互依赖性，同时，排序还可以节省用户的宝贵时间。

例如，如果先触发了访问令牌的验证操作，那么其他任务可能会并行执行，因为验证过程需要进行网络连接。但是这样就会导致一种情况：如果其他任务先完成，而验证还未完成，应用就必须等待验证完成才能继续执行。

(3) 将任务拆分为两类：一类是必须在主线程中执行的任务，另一类是可以在其他线程中执行的任务 ，然后分别执行。

还可以进一步将在非主线程中执行的任务分为可以并发执行的和不能并发执行的。

(4) 其他任务可以在加载 UI 后执行或异步执行。

延迟其他子系统（如记录仪和分析方法）的初始化。在应用的后续阶段将一些操作（例如，写日志消息或跟踪事件）放入队列中，直到子系统完全完成初始化。

你也许会注意到，其实并没有固定的解决方案。这些优化很大程度上依赖于对子系统的控制方式。针对崩溃上报、A/B 测试、埋点和分析、网络、图片缓存等子系统，有很多第三方解决方案。优化加载时间的最佳方式取决于选择的方法以及对它们有多少空间。

> 对于应用的依赖项，如果你能获取到代码，并且知道修复方法，那么请提交对应的补丁。通过对社区作贡献，你将帮助那些面临类似问题的人。
> 如果你购买了许可证，那么要积极联系该公司，你应该能得到答复。如果没有回应，那就毫不犹豫地抛弃它吧，寻找其他可替代的方法。最后要明确的是，是你的应用在与用户交互，第三方 SDK 在用户眼中并不重要。

举个例子，如果你发现分析 SDK 需要收集大量数据（如操作系统版本、应用版本、设备信息等）或需要从本地缓存加载一些配置，那么最好在非主线程异步进行初始化，同时将所有事件放入一个队列中（像 `NSMutableArray` 一样简单），在一次初始化中全部执行。

如果能获取到代码，打补丁就较为容易了。如果获取不到，你就需要维护自己的队列。这种做法的唯一缺点是，有些事件可能会出现不正确的时间戳和位置。在某些情况下，时间戳有几毫秒的偏差或定位有几米的偏差都是可接受的。但也有一些对准确度要求极高的情况（如果不准确，SDK 会表现很差），那可能就需要另做选择了。

例 5-2 中的代码提供了一种可以最大程度减少加载时间的方法。该示例列出了当代码不可用时，异步初始化分析 SDK 的必要步骤。

有些 SDK 需要在主线程中进行初始化。多留意这些情况，因为这会直接影响应用的加载时间。

例 5-2　应用加载时间的优化

```
//HPInstrumentation.m

@interface HPInstrumentation () ❶

@property (nonatomic, copy) BOOL initialized;
@property (nonatomic, strong) NSMutableArray *events;
@property (nonatomic, strong) dispatch_queue_t queue;

-(void)markInitialized;
+(void)logEventImpl:(NSString *)name;

@end

static HPInstrumentation *_instance; ❷

@implementation HPInstrumentation

+(HPInstrumentation *)sharedInstance { ❸
    return _instance;
}

+(void)setSharedInstance:(HPInstrumentation *)instance { ❹
    _instance = instance;
}

+(void)logEvent:(NSString *)name { ❺
    [[HPInstrumentation sharedInstance] logEventImpl:name];
}

-(instancetype)initWithAPIKey:(NSString *)apiKey { ❻
    if(self = [super init]) {
        self.initialized = NO;
        self.events = [NSMutableArray array];
        self.queue = dispatch_queue_create("com.m10v.queue.analytics",
            DISPATCH_QUEUE_CONCURRENT);

        dispatch_async( ❼
            dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
                [Flurry startSession:apiKey];
                dispatch_sync_barrier(self.queue, ^{ ❽
                    for(NSDictionary *name in self.events) {
                        [Flurry logEvent:name];
                    }
                    self.events = nil; ❾
                    self.initialized = YES;
                });
        });
    }
    return self;
}

-(void)logEventImpl:(NSString *)name { ❿
    dispatch_sync(self.queue, ^{ ⓫
```

```
        if(self.initialized) { ⓬
            [Flurry logEvent:name withParameters:params];
        } else {
            [self.events addObject:name];
        }
    });
}

@end

//HPAppDelegate.m
-(BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    HPInstrumentation *analytics = [[HPInstrumentation alloc]
        initWithAPIKey:@"API_KEY"]; ⓭

    [HPInstrumentation setSharedInstance:analytics]; ⓮
    [HPInstrumentation logEvent:@"App Launched"]; ⓯
}
```

❶ HPInstrumentation 是对底层埋点 API 的包装。它一直将事件保存于内存之中，直到底层 SDK（此处指的是 Flurry）被初始化。

❷ 因为在底层 SDK 准备好之前，我们需要在初始化时做一些调整，所以伪单例模式是首选。_instance 是可以被设置或重置的单例。[1]

❸ 共享实例 / 单例的获取方法。这是一个公共方法（在 .h 文件中声明）。

❹ 共享实例 / 单例的设置方法。这也是一个公共方法。
需要注意的是，如果任何代码都可以使用设置方法，那么可能会导致滥用。因此，必须谨慎使用。

❺ 设置日志事件的方法是公共的类方法，不会发生变化，而会使更新向后兼容。
至于实现，它采用了非公共的实例方法 logEventImpl。

❻ 类的自定义初始化器。

❼ 除了初始化状态，它调用 dispatch_async 来初始化底层 SDK[2]（此处指的是 Flurry SDK）。

❽ 一旦底层 SDK 被初始化，则刷新所有排队的 events。[3] 使用 queue 获取写入锁，从而确保当事件列表被刷新时，其他事件不被添加至列表中。

❾ 释放内存。

❿ logEventImpl 方法的实现。

⓫ 使用步骤 8 中同样的 queue，获取一个读取锁以保证并发写入。

⓬ 如果 SDK 尚未被初始化，则添加至事件列表，否则直接将日志写到底层 SDK。

⓭ HPInstrumentation 在应用委托中被实例化一次。

注 1：单例模式是指实体的一个实例，有多种实现方式。正如我们在第 2 章中讨论的，应尽量避免应用范围内不可复位的单例。

注 2：隐含假设——SDK 可以在非主线程中初始化。

注 3：另一个隐含假设——生成新事件的速度比刷新事件的速度要慢得多。如果不是，你有可能错误地使用了分析。再考虑考虑。

⓮ 设置共享实例。

⓯ 调用 `logEvent` 方法写日志。此处不需要任何改动。

注意，例 5-2 中的实现仅仅是多种备选方案之一。

使用委托进行初始化是一种更好的方式。初始委托添加至列表中，其他则使用实际的 SDK。例 5-3 提供了可用于初始化和切换的示例代码。

实现方式遵循状态设计模式。这种方法的优点在于可以很容易地管理底层实现。

例 5-3　使用委托进行初始化

```
-(instancetype)initWithAPIKey:(NSString *)apiKey {
    //……用于设置的相同代码

    self.delegate = [[HPInstrumentationUseList alloc] init]; ❶

    //以下代码用于初始化后
    dispatch_sync_barrier(self.queue, ^{
        for(NSDictionary *name in self.delegate.events) { ❷
            [Flurry logEvent:name];
        }
        self.delegate = [[HPInstrumentationUseSDK alloc] init]; ❸
    });
}

-(void)logEventImpl:(NSString *)name {
    dispatch_sync(self.queue, ^{
        [self.delegate logEvent:name]; ❹
    });
}
```

❶ 首先，委托指向一个对象，该对象将事件排入列表。

❷ 一旦准备完毕，刷新已经排列好的事件……

❸ ……同时改变委托的指向，将其指向使用分析 SDK 的对象。

❹ `logEventImpl` 更加简单，它使用委托记录日志。不再需要根据当前的状态（是否初始化成功）作出决定。

5.2.2　冷启动

上述的介绍应该让你对应用启动时执行的任务有了初步的了解。冷启动期间执行的任务只有很小的变化，却有可能产生很大的影响。

冷启动中一个较为重要的任务是，载入之前的状态。在应用中，显示给用户（登录后）的第一个画面是 feed 流。如果用户在以前的启动中登录过，并且数据已经同步，那我们就会考虑加载之前已经缓存的 feed 流。

我们将会在第 7 章和第 8 章深入讨论本地缓存的相关方法。此处先假设我们使用了一些方法，可以在记录表里做一些基本的 CRUD 操作。本节主要讨论如何较好地利用那些方法。

为了实现向用户展示 feed 流的任务，必须向服务器请求最近的更新，同时还要从本地缓存

加载数据。这些行为是不用思考就知道的。但是，以下几点却是不容忽视的。

- 展示有用且有意义的 UI 所需要的最少信息数目（min）。
- 记录从本地缓存加载 *M* 条信息花费的时间（记作 *tl*）。
- 记录从服务器获取最新的 *M* 条信息花费的时间（记作 *tr*）。
- 为了获得更快的速度，任何时刻在内存中存储的最大信息数目（max），特别是在快速滑动和滚动时。



如果不能在 3 秒内加载 *M* 条信息，那么用户体验将显著下降。[4]

这些值有助于为应用启动时的数据检索定义一个具体的策略。

考虑以下几种情况：

(1) *tl* = 3 秒，*tr* = 1 秒
(2) *tl* = 1.5 秒，*tr* = 1.5 秒
(3) *tl* = 1 秒，*tr* = 3 秒

假设 min=5，max=20。我们将针对这些时间对不同的 *M* 值进行讨论。

在我们的案例研究中，视图层级如图 5-5 所示，我们将按照此种情况进行时间测量。



图 5-5：视图控制器结构

注 4：J. O’Dell, VentureBeat, “This Is Why Users Think Your Mobile App Sucks: A 3-Second Response Time” (http://venturebeat.com/2013/09/02/this-is-why-users-think-your-mobile-app-sucks-a-3-second-response-time/).

注意，实际的容器层是 window。根视图控制器 HPMainTabBarViewController 包含了三个子控制器：HPChapterViewController、HPDebugLogViewController 和 HPSettingsViewController。图 5-5 只展示出了当前可见的视图控制器。子控制器也存在，只是不可见而已。

1. 情景1

如果从远程服务器同步数据比从本地缓存读取更快，那么你应该尽早地触发同步任务。在 Cocoa 推崇的典型 MVC 架构中，对应的 UIViewController 负责触发任务，因为视图控制器知道需要何种数据来填充 UI。一种可能的实现方式是，在应用的委托中创建服务，将服务注入视图控制器，然后由视图控制器触发服务。当然，如果想要节省额外的几毫秒，从应用委托触发同步也是没有问题的。创建视图控制器或只加载 shell UI 都需要花费一些时间。

如果以 ReactiveCocoa 框架为基础，真正实现起来应该不会太复杂。我们可以创建一个信号，将其传递给视图控制器，在需要时向下传递。一旦数据可用，视图控制器就会收到通知。此外，除了网络不可用的特殊情况，其他时候都可以正常地使用本地缓存。

在第一个分情景中，假设检索大量信息所需的时间能够完全达到甚至超出向用户提供较好用户体验这一目标所需的最小值。这意味着，一旦服务器数据可用，那么就无需其他操作。因为数据可以直接提供给应用并触发刷新。

例 5-4 展示了如何使用 Promise。Promise 开始于应用委托，但在视图控制器中使用。这种做法似乎违反了 MVC 的原则，所以很多人不赞同。换一种角度看，这其实只是一个依赖注入。Promise 是注入视图控制器的数据源。

例 5-4　使用数据源注入的视图控制器

```
//HPAppDelegate.m
-(BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    //验证访问令牌、用户登录等
    RACSignal *feedSignal = [[[HPSyncService sharedInstance]
            fetchType:@"feed"]] replay]; ❶

    HPUserFeedViewController *viewController =
        (HPUserFeedViewController *) self.window.rootViewController;

    viewController.feedSignal = feedSignal; ❷
}

//HPUserFeedViewController.m
-(void)viewDidLoad {
    @weakfily(self);
    [[self.feedSignal ❸
        deliverOn:[RACScheduler mainThreadScheduler]] ❹
        subscribeNext:^(HPUserFeed *feed) {
            @strongify(self);
            [self updateWithFeed:feed]; ❺
            self.feedSignal = nil; ❻
        } error:^(NSError *) {
            //处理错误
```

```
            }];
    }
```

❶ 创建信号。
❷ 假设 `rootViewController` 是 `HPUserFeedViewController`，包含一个 `feedSignal` 属性，并将其设置为先前创建的信号。
❸ 信号的订阅在控制器中执行。因为此操作是在 `viewDidLoad` 中完成的，所以可以保证只被执行一次。
❹ 响应的辅助调用在主线程中运行。
❺ 触发 UI 更新。
❻ 释放 `feedSignal`。这步操作会使引用计数减 1，最终会让它析构。

在其他分场景中，如果得到的信息数不足以提供良好的用户体验，那么你可以选择以下方法之一。

- 提高服务性能以获取足够的信息。
- 如果因为某种原因，服务不受控制，那么你需要寻找其他的解决方法。看看同样的内容是否能够以不同的方式重新渲染，如果新的渲染结果能占用更多的展示空间，那么就可以减少拉取的信息条数，也就不需要通过增加空白空间来填充界面了。

  例如，一张图片本来展示为小缩略图，但我们可以尝试将其放大，这样就能得到一张较好的概要图了。请牢记，这些新的设计实际上可以带来更好的用户体验。图 5-6 展示了 Facebook 和 Yahoo Finance 的屏幕截图。通过将左侧较旧的截图与右侧较新版本的截图进行比较，我们可以看出，在较新的截图中，单个条目不仅占用了更多的空间（信息密度低），而且在视觉上也更吸引人。

2. 情景2

如果从本地缓存加载的时间与从服务器中检索所需的时间相当，那么最好同时触发两者的操作。

举个例子，一个邮件应用可能需要一段时间才能打开，因为除了从本地缓存加载数据，它还必须从服务器同步数据。它从本地加载数据的时间很可能与获取新邮件的时间相当。

如之前所述，拉取应该尽早地触发。将信号注入视图控制器，并从那里获取数值。视图控制器需要同步更新。如果从服务器端获取的数据是可用的，那么本地缓存的数据应该丢弃，毕竟服务器端的数据更新一些。

有了这些改变，视图控制器的代码与例 5-5 中的代码类似。

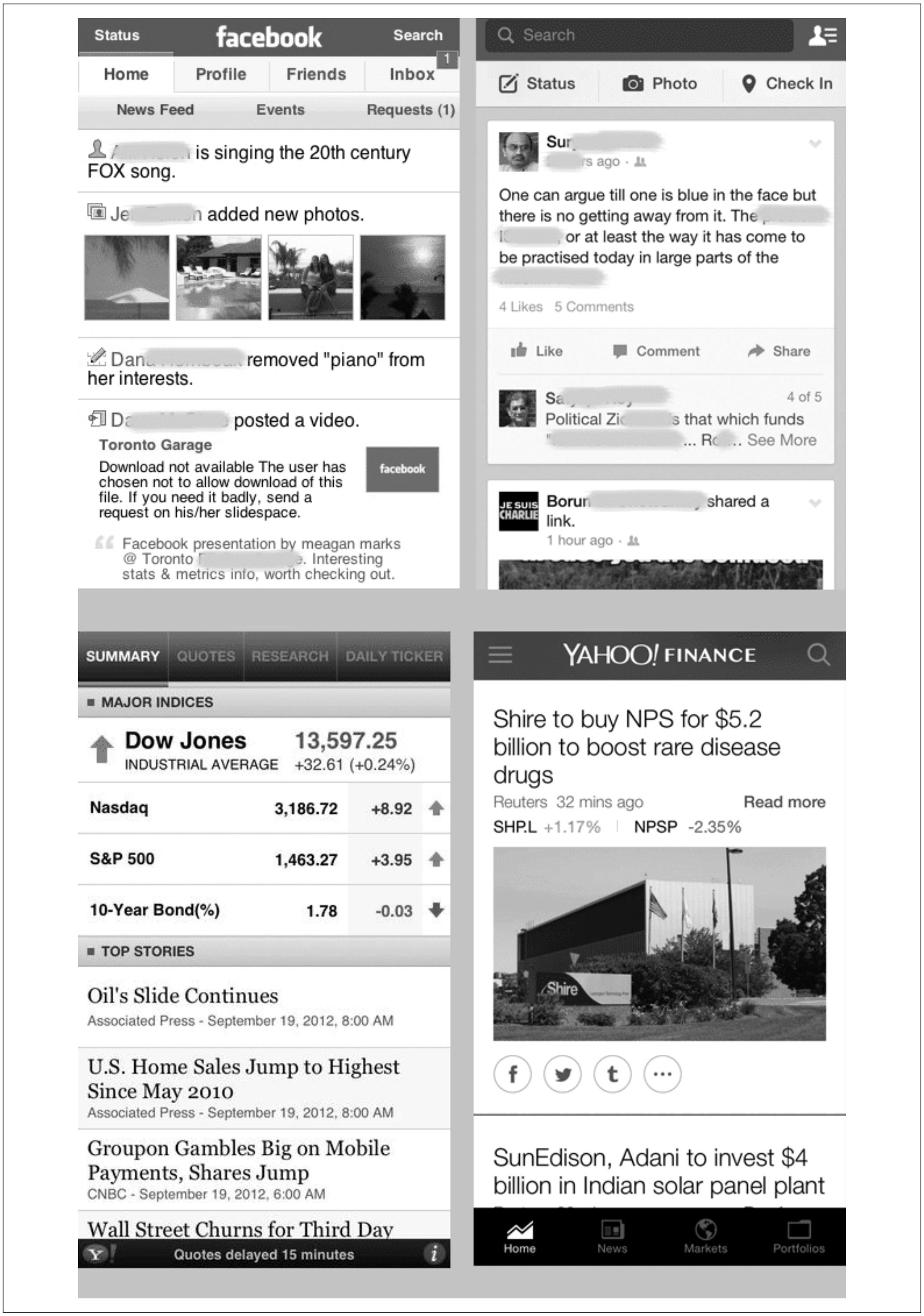


图 5-6：过去和现在的 Facebook 和 Yahoo Finance

例 5-5　有多种数据源注入的视图控制器

```
@interface HPUserFeedViewController ()

@property (nonatomic, copy) BOOL updatedFromServer; ❶

@end

@implementation HPUserFeedViewController

-(void)viewDidLoad {
    self.updatedFromServer = NO;

    @weakify(self);
    [[self.cacheFeedSignal ❷
        deliverOn: [RACScheduler mainThreadScheduler]
        subscribeNext:^(HPUserFeed *feed) {
            @strongify(self);
            [self updateWithFeed:feed fromServer:NO];
            self.cacheFeedSignal = nil;
        } error:^(NSError *error) {
            //处理错误
        }];

    [[self.serverFeedSignal ❸
        deliverOn: [RACScheduler mainThreadScheduler]
        subscribeNext:^(HPUserFeed *feed) {
            @strongify(self);
             [self updateWithFeed:feed fromServer:YES];
             self.cacheFeedSignal = nil;
         } error:^(NSError *error) {
             //处理错误
         }];
}

-(void)updateWithFeed:(HPUserFeed *)feed
    fromServer:(BOOL)fromServer { ❹

    if(self.updatedFromServer) { ❺
        return;
    }
    //刷新UI
    self.updatedFromServer = fromServer; ❻
}

@end
```

❶ `updatedFromServer` 是一个私有属性，用来追踪从远程服务器获取的数据是否引起了更新。
❷ 订阅 `cacheFeedSignal`，它可以接收来自本地缓存的数据。
❸ 订阅 `serverFeedSignal`，它可以接收来自远程服务器的数据。
❹ 更新 UI 的方法新增了一个额外的参数——区分数据源是否为远程服务器的标记。
❺ 如果 UI 已经从服务器接收到了更新数据，那就没有必要再进一步地更新了。当服务器的响应比从本地缓存加载更迅速时，就会出现这种情况。

❻ 一旦完成刷新，设置 `updatedFromServer` 标记，表明刷新是否使用了服务器的数据。

如果从某一单独的数据源获得的数据不足以提供良好的用户体验，最好的做法是结合两个数据源的结果，然后展示最终的结果。总之，检索结果所用的时间是相当的，结果的合并对用户而言也是透明的。

3. 情景3

最常见的情境是，从本地缓存获取数据所需的时间比从服务器上检索最新数据花费的时间更短。在这种情况下，应该先加载旧数据，最新的数据可用时再更新。

从实现的角度来看，解决方案类似于我们在情景 2 中所看到的——将两个数据源注入视图控制器中。唯一的区别是，在 `updateWithFeed:fromServer:` 方法中，条件语句 `if(self.updatedFromServer)` 被执行的概率几乎为零。

5.2.3 热启动

热启动是指切换到一个已经运行了的应用。两个原因可能会使应用变成非激活状态：一是用户向下拉拽状态栏，二是用户点击 home 键或切换至其他应用。

热启动有两种情境：

- 用户点击图标
- 应用接收到深层链接

1. 应用重启

当用户点击应用图标时，一般不需要执行其他特殊的操作。

应用处于安全状态，或者运行很多动画时，可以监测背景和前景通知。在第一种情况下，应用每次进入前景状态时，都会展示登录界面；在后一种情况下，动画或者游戏状态会被暂停，需要恢复。图 5-7 展示了 Temple Run 和 Intuit Mint 处理这种情景的方式。

此外，其他操作类似于用户和应用继续进行交互时的操作。

2. 深层链接

当应用接收到 `application:openURL:sourceApplication:annotation:` 回调时，期望能跳转到应用的特定页面，实现用户想要完成的操作。但此时的目标应用可能已经发生变化，处于某一特定状态了。

如果深层链接需要从服务器获取数据，那么可以先展示与深层链接相关的原始页面，或者先展示一个进度条，等从服务器获取到了最新数据，再执行刷新操作。

为了实现用户想要完成的操作，你可以遵循下列最佳实践。

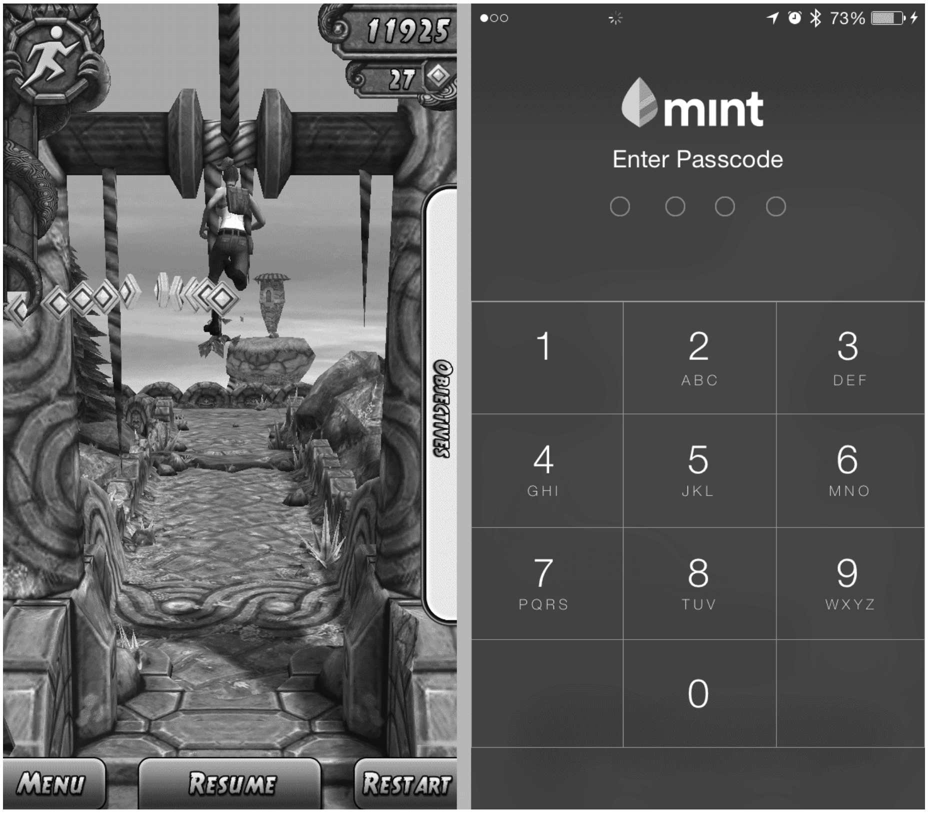


图 5-7：应用热重启——Temple Run 和 Mint（Temple Run 暂停了游戏并期望用户继续，Mint 用密码保证访问）

- 提供一个“返回”源应用的选项，通过该选项支持深层链接。最简单的方式是，当应用处理完相关事件后，在传入的 URL 上增加一个参数，然后将其当作最终的目的 URL。举个例子，Facebook 应用可以深层链接到 Messenger 应用。在链接时提供一个“后退”选项，一旦完成消息的相关操作，就可以直接回退至 Facebook 应用。
- 实现应用时，最好能将其当作一个简化的有限状态机。这样就可以推送新的屏幕，并在交互完成时直接弹出。
  如上述例子，如果打开 Messenger 应用，则其中没有返回到 Facebook 应用的“返回”按钮。此时，如果深层链接到 Messenger 应用，就像是直接“推送”到应用。

  当然，这里的目的是“与朋友聊天”。一旦目的实现，如果能直接返回到应用启动的地方，那么会显得更加友好一些。需要注意的是，在这种形式的通信中，两个应用都必须支持深层链接。

iPhone 手机现在仍然没有硬件后退按钮，这意味着应用必须适应 UI 本身。图 5-8 显示了

其他两个应用的例子：Chrome 浏览器和谷歌地图。注意，让每一个应用返回原应用的界面各不相同，且都高度依赖目标应用。

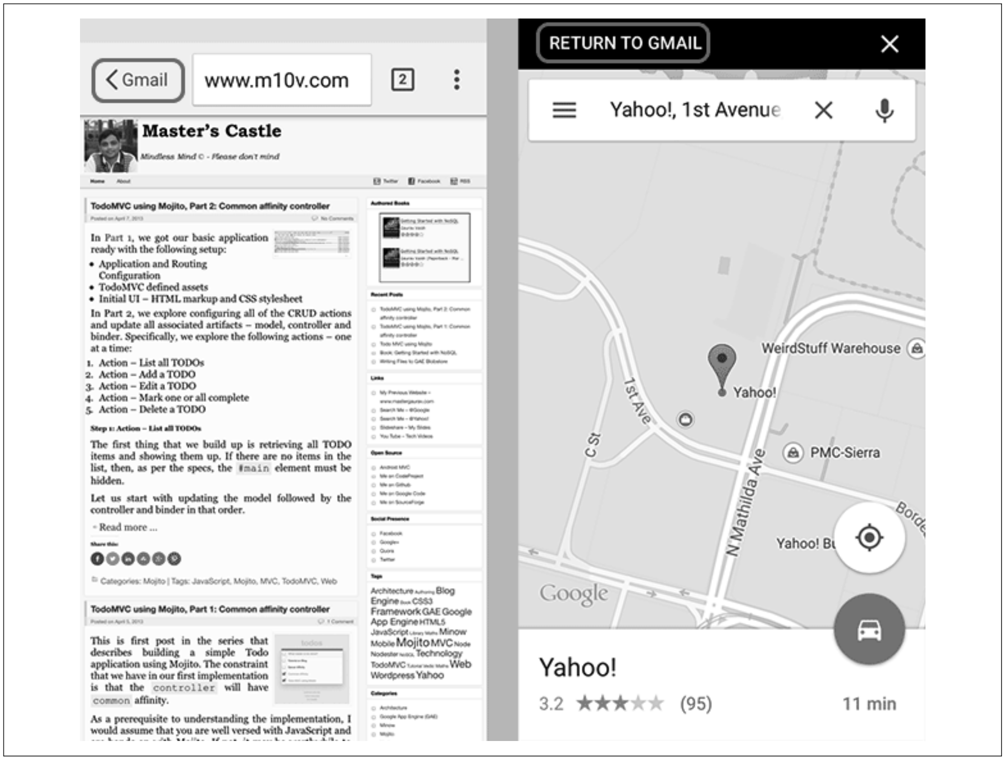


图 5-8：有“返回”导航支持的深层链接

5.2.4 升级后启动

应用升级后的首次启动将遵循下列情形之一：

- 无本地缓存或应用完全放弃缓存；
- 本地缓存可用，可以直接使用或需要切换至升级版本。

如果无本地缓存或应用决定放弃缓存（例如，数据不可用或从服务器同步获取更快），则不需要进行特殊处理。

本地数据发生改变时通知用户。以下的最佳实践可以让用户有更好的体验。

- 如果本地缓存可用，通知用户该情况。如果没有迁移到本地缓存的必要，则无需通知用户，因为本地缓存的使用是隐式的。
- 如果必须花几分钟对数据进行迁移，那么向用户展示一个可以推迟该操作的选项。
- 如果从服务器检索数据更快、更容易，因而必须放弃本地缓存的使用，那么这种情况下需要通知用户。

有个现成的邮件应用的例子，将旧版本的记录迁移至新版本的升级模式时所做的工作，要比从服务器上直接检索数据更加复杂。

5.3 推送通知

通知是内容驱动型应用的一个组成部分。这里的内容可以像新闻一样组合，也可以像邮件一样由用户生成。实现这些的前提条件是，你了解应用委托中的 `application:didReceiveRemoteNotification:` 和 `application:didReceiveLocalNotification:` 回调。

同时，你还需要了解调用方法的顺序，以及用户交互是如何驱动其他方法实现调用的。这两方面非常重要，因为它们会影响应用初始化时遵循的顺序，以及特定回调中执行的子组件初始化的种类。在特定回调中执行不同子组件的初始化，可以实现资源利用的最小化，最大化要完成的任务的数量。

5.3.1 远程通知

图 5-9 显示了收到通知时的委托回调。

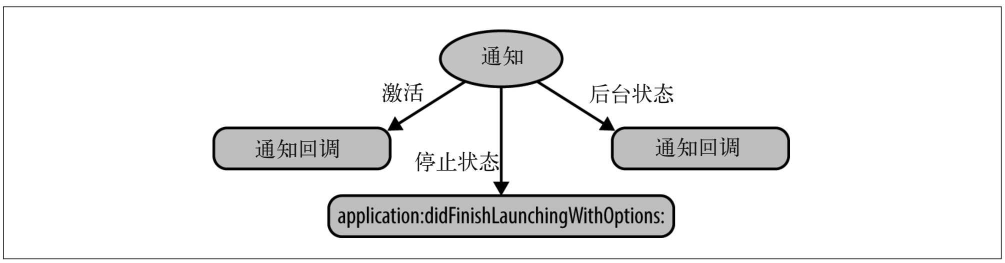


图 5-9：应用委托中关于通知的完整生命周期

以下是对 iOS 8 生命周期的描述。

- 如果应用是激活状态的，则通过 `didReceiveRemoteNotification` 回调接收通知。
  没有调用其他回调，为避免分心，也没有向用户展示 UI。
- 如果应用在后台运行或停止，只有静默推送通知回调会被触发。
  基于通知设置，非静默推送通知可能会出现在通知中心或作为报警弹窗，或更新应用图标的角标计数。
- 当用户使用通知中心或报警弹窗开启通知时，可能会发生以下情况中的一种。
  - 如果应用处于后台，则通知回调方法会被调用。
  - 如果应用处于停止状态，则 `application:didFinishLaunchingWithOptions:` 方法的 `launchOptions` 参数中的通知对象（`NSDictionary`）是可用的。

你可能已经注意到，生命周期并不明确，它高度依赖于应用的状态，这就使得代码中的多个地方都需要处理程序块。

用于处理通知的典型方式与例 5-6 中所示代码类似。

例 5-6　处理通知

```
-(void)application:(UIApplication *)application
    didReceiveRemoteNotification:(NSDictionary *)userInfo { ❶
    //查看下一个方法
}

- (void)application:(UIApplication *)application
    didReceiveRemoteNotification:(NSDictionary *)userInfo
    fetchCompletionHandler:(void (^)(UIBackgroundFetchResult))completionHandler {❷
    //处理远程通知—应用正在运行 ❸
    if(application.applicationState == UIApplicationStateInactive) { ❹
        //用户点击通知中心的通知或报警弹窗
        [self processRemoteNotification:userInfo];
    } else if(application.applicationState == UIApplicationStateBackground) {
        //应用在后台,不存在用户交互——只是获取数据
    } else {
        //应用已经处于激活状态—显示应用内的更新
    }
}

-(void)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions { ❺
    id notification = [launchOptions
        objectForKey:UIApplicationLaunchOptionsRemoteNotificationKey];

    if(notification != nil) { ❻
        NSDictionary *userInfo = (NSDictionary *)notification; ❼
        [self processRemoteNotification:userInfo];
    }
}

-(void)processRemoteNotification:(NSDictionary *)userInfo { ❽
}
```

❶ iOS 7 回调。

❷ iOS 8 回调。如果实现了回调，它将取代 `application:didReceiveRemoteNotification` 方法。

❸ 此回调只在应用运行时才会被调用，否则该回调不会被触发。

❹ 如果应用处于前台，最好不要立即切换 UI，首选方案是在应用中展示一个恰当的提示（如应用内横幅）。应用处于后台时，用户点击通知后会触发回调，此时便可随意切换 UI 了。

❺ 所有的“魔术”都藏在这里。

❻ 检查 `UIApplicationLaunchOptionsRemoteNotificationKey` 键是否存在。

❼ 如果存在，则远程通知数据是有效的。然后处理这些数据。

❽ 处理通知的核心位置。

以下是管理生命周期时可以遵循的一些最佳实践，有助于应用提供最佳的用户体验。

- 当应用在活动状态时收到一条通知,要么会忽略该通知,要么会显示恰当的提示。举例如下。

- ♦ 对邮件应用来说，如果收到有关新邮件的通知，更新邮件文件夹的角标即可。如果接收新邮件通知时，线程当前属于开启状态，较好的选择是更新线程并显示一个指示符表明新邮件可用。同样，如果收到的是不同的聊天会话新消息，你或许可以显示一个应用内横幅，如图 5-10 所示。横幅位置应保证不会妨碍用户继续完成当前任务。
- ♦ 对消息应用来说，如果在当前开启的聊天会话中收到新信息通知，直接在内部展示信息，并更新线程。（不要对所有信息都使用推送通知，因为不能保证 100% 传送。）

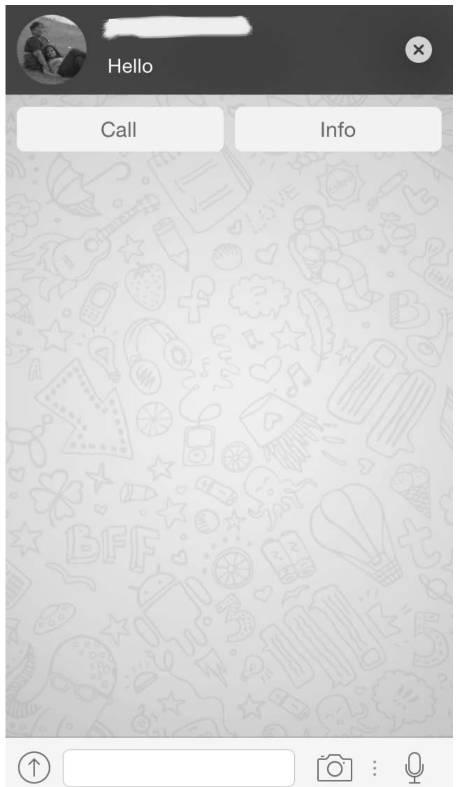


图 5-10：在不同的聊天会话中收到新消息时，WhatsApp 显示应用内横幅

- 当应用在非活动状态时收到通知，一定是因为用户之前点击了通知。但此时用户可能已经在做别的事情了，这种情况下，比较好的办法是推送一个新页面，这个页面上有支持用户直接返回之前页面的选项。用户体验的准确定义取决于应用和行为。
  以某一金融应用为例，如果当前屏幕正在显示账户概览，却收到一个有关刚完成的交易的通知，此时可以向用户显示通知的相应细节，并提供“返回”导航让用户直接回退至账户概览。
- 当应用在 `application:didFinishLaunchingWithOptions:` 回调中获取到通知对象的详细信息时，直接将通知相关的 UI 展示给用户。

需要注意的是，`application:didReceiveRemoteNotification:` 只有在应用处于前台时才会被调用，然而，如果实现了 `application:didReceiveRemoteNotification:fetchCompletionHandler:`，当应用处于后台或还没有运行时，此回调也会被触发。这也就是说，此回调甚至会启动应用。这样的通知被称为*静默推送通知*。

同样需要引起重视的是，后一种方法可能被调用两次。

- 第一次是收到通知，并且 payload 字段包含一个键为 `content- available`、值为 `1` 的键值对时。[5]
- 第二次是用户以通知中心或警告的方式和通知交互时。

可以使用应用的状态来区分两者，如例 5-6 所示。

通知应该是有意义的。payload 字段应包含展示给用户的文本信息以及一些数据，这些数据在回调中可用来确定是否需要触发后台拉取。

5.3.2 本地通知

不同于远程通知，当应用处于使用状态时，本地通知不会展示任何 UI。

因此会产生如下问题。

- 如果应用处于使用状态，为什么你还需要显示本地通知?
- 如果应用没有处于使用状态，它被挂起了，在这种情况下怎样显示本地通知?

答案便是*静默远程通知*。如果远程通知 payload 字段的 `content-available` 属性值被置为 `1`，这表明它会告诉操作系统远程通知不应展示给用户，而是必须直接传递给应用。与普通的推送通知类似，如果需要的话，这有可能会唤醒应用。

随后，应用可以处理数据，需要时还会触发远程拉取，同时创建一个本地通知。这种方法的优点是，当用户与通知交互时，数据可能已经被下载、处理并以可用的形式提供给应用。如此一来，显示通知细节的时间会变得较短，应用响应会加速，这更能取悦用户。

当应用处于前台，或应用处于后台，但用户点击了通知时，`application:didReceiveRemoteNotification:fetchCompletionHandler:` 回调（如果没有实现，则调用 `application:didReceiveLocalNotification:` 回调）会被调用。但是当用户在通知中使用自定义行为时，`application:handleActionWithIdentifier:forRemoteNotification:completionHandler:` 回调会被调用。

> 同时使用本地通知与静默推送通知，可以使得应用在下次启动时响应和使用速度都更快。

注 5：Stack Overflow,“didReceiveRemoteNotification:fetchCompletionHandler Not Being Called When App Is in Background and Not Connected to Xcode”(http://stackoverflow.com/questions/20741618/didreceiveremotenotificationfetchcompletionhandler-not-being-called-when-app-is/20851481#20851481).

5.4　后台拉取

后台拉取功能是在 iOS 7 中推出的，该功能可以较好地从服务器定期同步数据。要想启用后台拉取功能，需要以下 3 个基本步骤。

(1) 在项目设置中开启功能。

(2) 设置刷新间隔，最好在 `application:didFinishLaunchingWithOptions` 中完成。使用 `-UIApplication setMinimumBackgroundFetchInterval:` 方法请求刷新以指定的频率完成。

(3) 实现应用的 `application:performFetchWithCompletionHandler:` 委托方法。如果任务没有在 30 秒内完成，操作系统会调度执行频率较低的方法去运行。

实践记录表明，应用一般使用的时间要少得多，通常在 2~4 秒。苹果公司的开发者网站将 30 秒作为上限。[6]

后台拉取和推送通知可以为用户创造惊人的、愉快的体验。以下列举了一些可以创造深刻印象的指导原则。

- 使用后台拉取与服务器进行定期数据同步。将它当作你想要执行的批量操作。
- 不要过分依赖后台任务执行的规律性。操作系统会定期调度它，但是调度的时间间隔却是无规律的。

  白天，这种间隔通常在 10~20 分钟（参见图 5-11）。影响数值变化的因素有：`UIBackgroundFetchResultNoData` 或 `UIBackgroundFetchResultNewData` 被当作最终响应结果的频率、完成操作所花费的平均时间、网络情况、预估的可用带宽、CPU 和可用内存等）。在夜晚，间隔可能会增加至几个小时。

```
15:43:14.388 HPerf Apps[20551:8463048] [W] [performFetchWithCompletionHandler] called
15:43:14.389 HPerf Apps[20551:8463048] <HPInst> App_Fetch
15:43:14.703 HPerf Apps[20551:8463048] [W] [shouldSaveApplicationState] called
15:43:14.705 HPerf Apps[20551:8463048] [W] [willEncodeRestorableStateWithCoder] called
16:03:38.540 HPerf Apps[20551:8463048] [W] [performFetchWithCompletionHandler] called
16:03:38.541 HPerf Apps[20551:8463048] <HPInst> App_Fetch
16:03:38.769 HPerf Apps[20551:8463048] [W] [shouldSaveApplicationState] called
16:03:38.771 HPerf Apps[20551:8463048] [W] [willEncodeRestorableStateWithCoder] called
16:13:14.042 HPerf Apps[20551:8463048] [W] [performFetchWithCompletionHandler] called
16:13:14.043 HPerf Apps[20551:8463048] <HPInst> App_Fetch
16:13:14.369 HPerf Apps[20551:8463048] [W] [shouldSaveApplicationState] called
16:13:14.370 HPerf Apps[20551:8463048] [W] [willEncodeRestorableStateWithCoder] called
16:28:14.453 HPerf Apps[20551:8463048] [W] [performFetchWithCompletionHandler] called
16:28:14.454 HPerf Apps[20551:8463048] <HPInst> App_Fetch
16:28:14.726 HPerf Apps[20551:8463048] [W] [shouldSaveApplicationState] called
16:28:14.759 HPerf Apps[20551:8463048] [W] [willEncodeRestorableStateWithCoder] called
```

图 5-11：后台拉取间隔的追踪（红色高亮部分）

- 使用推送通知唤醒或启动应用。
- 使用 payload 字段中的 `content-available = 1`，这样通知处理方法还可以从服务器同步数据。

  尽量只同步与通知项相关的项目，而不是同步所有的东西。

注 6：iOS Developer Library, “UIApplicationDelegate” (http://apple.co/1eMyYY0).

因为后台拉取使应用脱离了挂起状态，所以任何挂起的队列都可能恢复。如果应用的其他层没有意识到任务队列可能很快又被暂停，这也许会导致灾难性的崩溃。

使用 `NSNotifications` 通知应用的不同组件终止任何正在进行的操作，因为应用已经被唤醒来执行后台拉取操作，并且不久以后将再次暂停（30 秒为上限）。

使用 `backgroundSessionConfigurationWithIdentifier` 配置 `NSURLSessionConfiguration` 对象，从而生成 `NSURLSession`，这种做法有额外的好处。它可以使用操作系统级的守护进程来管理那些需要在进程外长时间运行的任务。当应用关闭或崩溃以后，后台网络会话还可以继续。

对非网络的操作而言，你需要自行实现类似的系统。

智能静默通知

我曾经参与开发的一款应用有很强的安全性要求。具体来说，登录成功后生成的访问令牌有 24 小时的有效期。在 24 小时之内，应用可以自动登录，无需用户重新输入凭据。但在 24 小时后，会话将过期，需要重新创建才行 。毋庸置疑，这必然不能带来很好的用户体验。

为了解决这个问题，我们使用后台拉取来刷新会话。

然而，我们发现，添加了这个功能以后，应用崩溃得更加频繁了。调查显示，挂起的操作队列恢复时并不知道应用很快会被再次挂起。其中有一个操作是与服务器同步数据，但因为被挂起了，所以同步无法完成，奇怪的事情发生了：由于没有活动，连接超时了。如此下去，服务器很少能接收到完整的数据；就算服务器有数据回应，客户端也很少能处理完整的数据，这就使得应用处于不一致的状态。

我们必须修复应用的同步层，确保它可以在触发同步操作之前得到应用的状态。

再来讨论静默通知。服务器在晚上会发送静默推送通知。这会唤醒应用，然后检查设备是否连接了 WiFi，是否有足够的电池（如果没有，是否在充电过程中）。如果上述条件都满足，则抢先执行会话刷新。

因此，无论在第二天何时与应用交互，用户都会觉得很顺畅。

5.5 小结

本章介绍了应用的生命周期，讲述了生命周期如何影响用户对应用的感知。要想创建出用户喜欢的应用，了解这些是至关重要的。

有时，实际性能不如感知性能重要。为了使用户对应用有良好的感知，聪明的做法是使用静默通知和后台拉取对应用进行热启动，这样可以为下次使用提前做好准备。

至此，本章已经介绍了各种情景下的优化技术，其中包括首次启动、冷启动、热启动以及升级后启动。相信你应该能够做一些改动，让应用的启动时间变得更短，使用起来更加顺畅。

第6章

用户界面

解铃还须系铃人。

——佚名

当与 UI 进行交互时，大部分用户才注意到性能问题。如果某个应用在数据同步和刷新上耗时较长，或用户交互不够稳定，那么应用会被认为是迟钝的。

功耗、网络使用率、本地存储等因素对用户来说是不可见的。因此，虽然这些因素是解决性能问题的要素，但 UI 却是应用的门面，如果 UI 反应迟钝，则必然会直接影响用户的反馈。

还有一些无法控制的外部因素，如下。

- 网络
  弱网环境会增加同步所需的时间。
- 硬件
  硬件越好，其提供的性能越高。与旧型号的 iPhone 相比，搭载新系统的新 iPhone 执行速度更快。应用可以在不同的CPU上运行，这些CPU包括了32位1.3GHz到64位1.8GHz的所有型号，同时支持 1GB 到 2GB 的 RAM。
- 存储
  应用可以在存储容量不同的设备上运行，存储容量小至 16GB，大到 128GB，它们限制了应用在本地离线缓存数据的规模。

应用还可以根据所处的运行环境作出不同的决策，保证用户交互的流畅性。

本章主要讨论如何最小化更新 UI 所需的时间。阅读完本章后，你应该可以找到对应的方法，让自己的应用以 60 帧每秒的帧率（fps）运行。这意味着应用会有 16.666 毫秒来完成

向下一帧过渡的全部操作。如果执行一条指令需要 1×10^{-9} 秒，应用在上述时间段大约可以执行 1000 万条指令。换一个角度来看，如果调用一个简单的、没有任何操作的方法需要大约 30 纳秒（包括设置栈帧、参数入栈、执行以及最终清理的时间），那么执行 50 多万个方法也是绰绰有余的。所以在这个时间段能够执行很多方法。

> **帧　　率**
>
> 人的眼睛、大脑界面以及视觉系统每秒可以处理 10~12 个独立的图像，并且可以单独感知它们。人类视觉感知的阈值取决于被测物。
>
> 当看向较为明亮的显示器时，如果一处黑暗持续了 16 毫秒或更长时间，人们就会开始关注它。
>
> 根据设备的性能，有一些方法可以优化帧率。例如，如果要在一个 RAM 较小的设备上运行应用，那就在内存中载入较少的数据。另一个例子是，在较慢的 CPU 上运行时，尽量减少动画的使用。

本章将关注以下部分：

- 视图控制器及其生命周期
- 视图渲染
- 自定义视图
- 布局
- 应用扩展（widgets）
- 动画
- 交互式通知

我们将深入研究这些组成部分，探索可以优化执行的方式，并学习一些技巧，以便用户能够获得较好的感知体验。

6.1　视图控制器

视图控制器就像胶水一样连接着数据服务和视图。数据服务不仅对内存数据提供支持，还可以从服务器或者本地数据库请求或推送更新。

视图控制器的生命周期依赖于它的 `view` 属性，该属性决定了视图被创建、展示、删除和销毁的时刻。图 6-1 展示了视图控制器的生命周期。

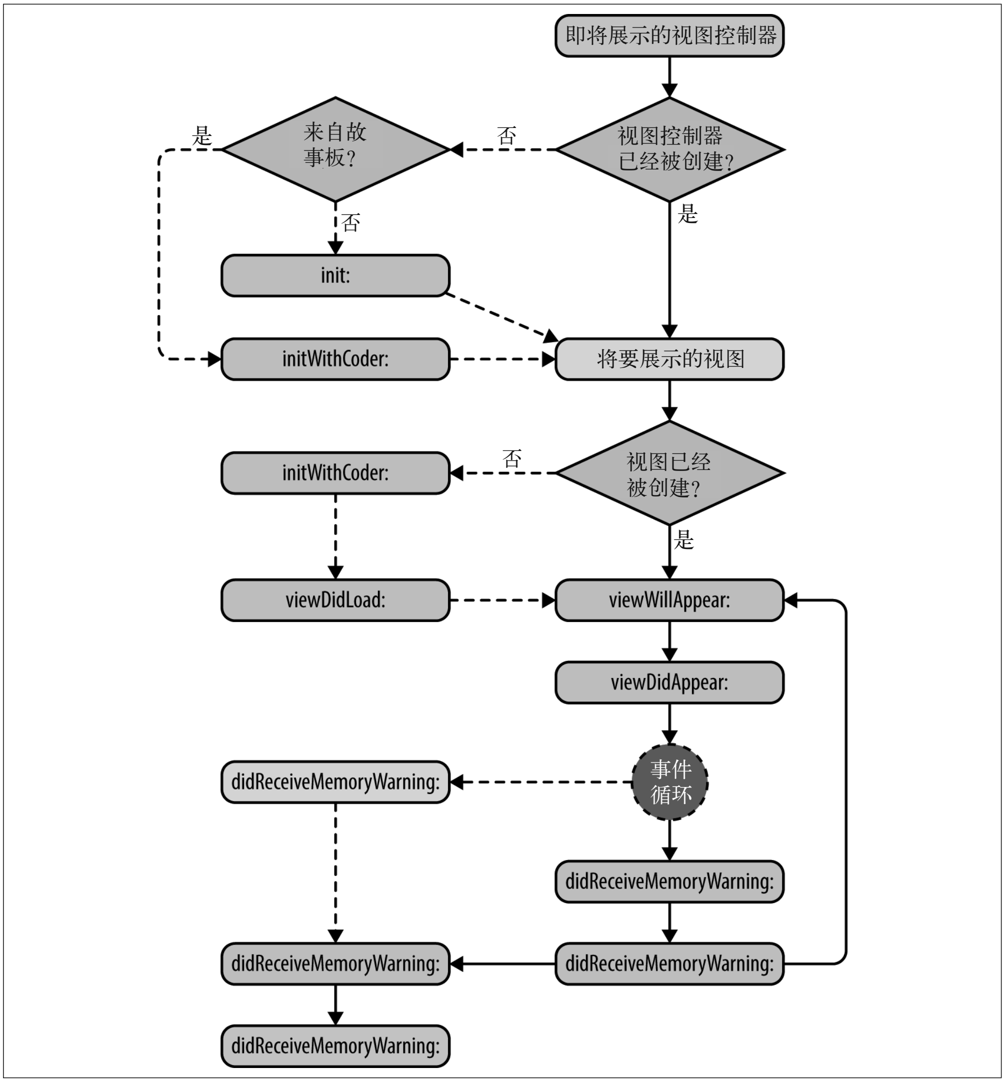


图 6-1：视图控制器的生命周期

在应用开发的最初阶段，视图控制器都较为精简，状态较好。随着时间的推移，这些视图控制器慢慢变成了所有业务逻辑的垃圾场，代码量也增长至几千行。虽然逻辑的“总量”是不可避免的，但将代码重构成短小、可复用的方法是很好的主意。这样不仅能解除耦合，还可以发现无用的、重复的代码。

下面列举了创建视图控制器时需要遵循的一些较为基本的最佳实践。

• 保持视图控制器轻量。在 MVC 结构的应用中，控制器只是纽带，而不是存放所有业务逻辑的地方。它甚至不属于模型。业务逻辑应该属于服务层或业务逻辑组件。将它放在那里。

通过被称为行为委托或服务提供者的对象，视图控制器应该将服务组件绑定至视图，这些行为委托或服务提供者的对象最好能被注入控制器（参见 2.13 节）。

- 不要在视图控制器中编写动画逻辑。动画可以在独立的动画类中实现，该类接受视图作为参数传入，这些视图就是用来运行动画的视图。然后，视图控制器会将动画添加至视图或转场效果上。
  有特殊用途的视图可以拥有自己的动画。例如，一个自定义的微调控制器会有它自己的动画。
- 使用数据源和委托协议，将代码按照数据检索、数据更新和其他的业务逻辑进行分离。视图控制器只能用来选择正确的视图，并将它们连接到供应源。
- 视图控制器响应来自视图的事件，如按钮点击事件或列表单元格的选择事件，然后将它们连接至数据接收器。
- 视图控制器响应来自操作系统的 UI 相关事件，如方向变化或低内存警告。这可能会触发视图的重新布局。
- 不要编写自定义的 `init` 代码。 为什么呢？因为如果视图控制器被重新切换至 XIB 或故事板，那 `init` 方法永远都不会被调用。
- 不要在视图控制器中使用代码手工布局 UI，也不要在视图控制器中实现全部的 UI、视图创建和视图布局逻辑等操作。使用 nibs 或者故事板。
  手工布局代码不会持续很久，因为应用在不断增长，并且设计也在改变。在重新设计方面，使用 Interface Builder 比根据像素坐标来手动编写代码更快。

  此外，应用可能会在不同大小和形状的设备上运行。要想适应所有的形状，处理方向变化时的旋转操作，以及与每两三年就会变化的设计范例保持一致，通过扩展自定义代码来实现是比较难做到的。

  同样，如果某一设计被分在独立的 nibs 和故事板，你就可以比较灵活地运行 A/B 测试，因为在不同的约束之间很容易选择最终需要的。

- 比较好的方式是，创建一个实现了公共设置的基类视图控制器，其他视图控制器从这里继承就好。
  这种技术并非一直可用，因为有时可能需要在应用的不同部分继承不同的视图控制器。例如，在联系人列表中应该使用 `UITableViewController`，在用户配置文件中应该选择 `UIViewController`。

  但是，如果有多个地方都需要在 `UIWebView` 中展示内容，那基类视图控制器会是一个不错的选择。如果需要显示含有隐私策略的 URL 或条款和条件页面，那么你是没必要继承的。但是，如果需要显示用户分享的图片或者视频（在信息应用中），你可以创建子类，在子类中实现自定义浏览器或对重写的东西进行控制。

- 在各视图控制器之间，使用 category 创建可复用的代码。如果父视图控制器不能满足使用（例如，在应用中需要不同种类的视图控制器），那就创建 category，并在 category 中加上自定义的方法或属性。
  如此一来，你就不会被限制只能使用预定义的基类，同时还能得到复用带来的好处。

现在我们已经了解了编写视图控制器时的一些最佳实践。接下来，就 UIViewController 生命周期的方法而言，我们将探讨哪些事情是该做的，哪些事情是不该做的。

6.1.1 视图加载

视图初始化时会涉及两个方法——loadView 和 viewDidLoad。

不知你是否还记得，当添加一个新的视图控制器时，通过 Xcode 生成的模板代码只有 viewDidLoad 方法。当视图控制器的 view 被请求时，loadView 方法会被调用，但因为它还未被创建，所以会是 nil。

视图会通过以下三种方式加载：

- 从 nibs
- 使用故事板（使用 UIStoryboardSegue）
- 使用自定义代码创建 UI

如果通过覆写 loadView 方法创建了自定义 UI，你需要牢记以下几点。

- 将 view 属性设置到视图层级的根上。
- 确保视图正被其他的视图控制器所共享。
- 不要调用 [super loadView]。

使用此方法更改视图状态，尤其当视图是从 nib 文件加载而来时。我们主要讨论 viewDidLoad。

在视图层次结构准备就绪之后，视图呈现给用户之前，viewDidLoad 会被调用一次。可以在该方法中做一些一次性的初始化操作。

需要在 viewDidLoad 方法中做的公共任务包含以下几项。

- 配置数据源来填充数据。
- 为视图绑定数据。
  这是一个值得商榷的任务。根据不同的使用情况，你可以一次绑定数据，并提供一个刷新按钮，也可以每次调用 viewWillAppear 时绑定。使用后者的好处是，UI 总是展示最新的数据；缺点是，如果数据更新不频繁（例如，在新闻应用中），用户每次看到的都是不必要的刷新（例如，当 UITableView 重新绑定时）。
- 绑定视图的事件处理程序、数据源的委托和其他回调。
- 注册数据的观察者。
  依据数据绑定到视图时的不同位置，数据的观察者可能也会发生变化。
- 从通知中心对通知进行监控。
- 初始化动画。

在执行过程中，应该尽量缩短在 viewDidLoad 方法上花费的时间。具体来讲，将要被渲染的数据应该是已经可用的，或是在其他线程进行加载的。在 viewDidLoad 的完成中发生的任何延迟，都将导致与视图控制器相关的 UI 展示发生延迟。用户会卡在应用启动或前一个视图控制器中。

6.1.2 视图层级

展示出来的 UI 是由嵌套在树形结构中的各层次视图组成的，它们的位置受自动布局或其他编排方式的约束。视图结构和渲染包括以下步骤。

(1) 构造子视图。
(2) 计算并提供约束。
(3) 为子视图递归地执行步骤 1 和步骤 2。
(4) 递归渲染。

由于视图层次变得复杂，因此需要更长的时间来构建和渲染视图。考虑一个简单的平面层次结构：

- `UILabel`
- 自定义视图
- `UIImageView`
- `UILabel`

在一个安装了 iOS 8.1 的 iPhone 6 上，从视图控制器加载（`initWithCoder:`）到渲染（`viewWillAppear:`）之前的平均耗时约为 15 毫秒。而且，这还没有考虑从磁盘（故事板 / nib 文件）一次性加载布局的时间。`viewDidAppear`：方法会因为过渡动画的原因在约 300 毫秒后被调用。

图 6-2 和图 6-3 分别显示了视图层级和记录日志的时间。

图 6-2 所示的简单 UI 花费了约 15 毫秒来加载，留给其他操作约 1.6 毫秒，这样才能实现 60 帧每秒的渲染。由于 UI 变得越来越复杂，并需要处理更多的数据，因此优化执行变得越来越重要。

如果一个简单的示例需要花费如此长的时间来加载（不渲染）UI，只留下大约 1 毫秒来完成所有其他的操作，那么我们可以推断出，为了实现 60 帧每秒的渲染，所有操作必须在 16.66 毫秒内完成。

帧率与丢帧

因为 nib/ 故事板加载的时间和构建视图的时间并没有可以优化的空间，所以你需要回头看看绘图板，找出让用户感到不满的因素。这是因为应用不能以 60 帧每秒的速率运行还是因为应用有一些跳变？

如果应用每秒丢 1 帧，并以 59 帧每秒的速率运行，那么还是顺畅的。但如果前 5 秒都是以 60 帧每秒的速率运行，而在第 6 秒丢掉了 5 帧，那用户肯定会注意到此处发生了跳变。

虽然已经过去几年了，且 Andrew Munn 的文献 Follow up to “Android Graphics True Facts,” 或 The Reason Android is Laggy（https://plus.google.com/u/0/+AndrewMunn/posts/VDkV9XaJRGS）大都是关于 Android 的，但现在还是值得在睡前阅读一下。

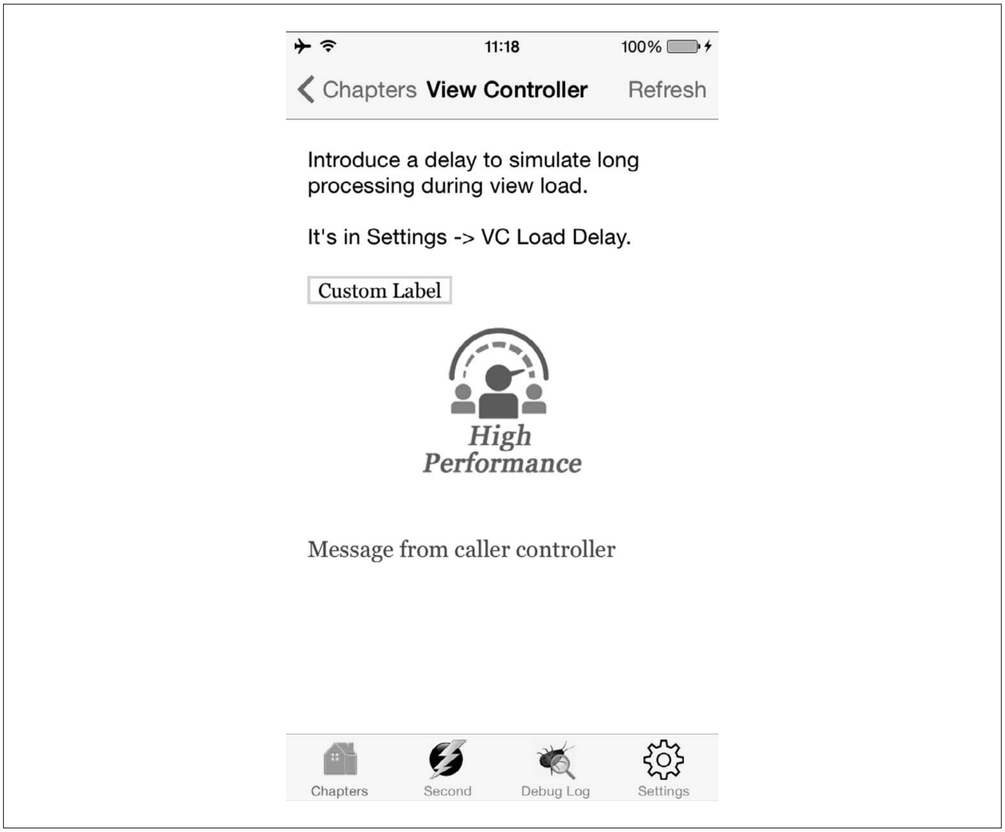


图 6-2：视图层次

```
-14 11:28:08.570 HPerf Apps[1788:541924] [I] [VC::initWithCoder]: 0.000410
-14 11:28:08.571 HPerf Apps[1788:541924] [V] prepareForSegue[HPCVC] destination: ident=segue_ch07_al_vc cls=HPChap
-14 11:28:08.572 HPerf Apps[1788:541924] [I] [VC::setMessage]
-14 11:28:08.586 HPerf Apps[1788:541924] [I] [MV::didAddSubview] time=8750ns for UILabel
-14 11:28:08.588 HPerf Apps[1788:541924] [I] [MV::didAddSubview] time=1000ns for HPCustomLabel
-14 11:28:08.590 HPerf Apps[1788:541924] [I] [MV::didAddSubview] time=750ns for UIImageView
-14 11:28:08.590 HPerf Apps[1788:541924] [I] [MV::didAddSubview] time=583ns for UILabel
-14 11:28:08.591 HPerf Apps[1788:541924] [I] [MV::didAddSubview] time=416ns for _UILayoutGuide
-14 11:28:08.592 HPerf Apps[1788:541924] [I] [MV::didAddSubview] time=791ns for _UILayoutGuide
-14 11:28:08.594 HPerf Apps[1788:541924] [I] [VC::p:viewDidLoad]: 0.024300
-14 11:28:08.595 HPerf Apps[1788:541924] [I] [VC::viewDidLoad]: 0.025468
-14 11:28:08.595 HPerf Apps[1788:541924] [I] [VC::viewWillAppear]: 0.025855
-14 11:28:08.596 HPerf Apps[1788:541924] <HPInst> SCR_ViewController
-14 11:28:08.611 HPerf Apps[1788:541924] [I] [MV::layoutSubviews] time=2420416ns
-14 11:28:08.615 HPerf Apps[1788:541924] [I] [MV::drawRect] time=1083ns
-14 11:28:09.124 HPerf Apps[1788:541924] [I] [VC::viewDidAppear]: 0.554471
-14 11:28:09.126 HPerf Apps[1788:541924] [I] [MV::layoutSubviews] time=74250ns
```

图 6-3：记录时间

现在我们的重点不仅仅是尽量在 16 毫秒内完成主线程的任务执行，同时还要最大限度地减少丢帧数（更明确来说是大量丢帧）。

6.1.3　视图可见性

视图控制器提供了四个生命周期方法，以接收有关视图可视性的通知。

- `viewWillAppear:`

  当视图层级已经准备好，且视图即将被放入视图窗口时，此方法会被调用。在即将展示视图控制器或之前入栈（modal 或者其他）的视图控制器弹出时，这种情况就会发生。

- 在这个时刻，过渡动画还未开始，视图对终端用户也是不可见的。不要启动任何视图动画，因为没有任何作用。

- `viewDidAppear:`

  当视图在视图窗口展示出来，且过渡动画完成后，此方法会被调用。

  因为动画会耗费约 300 毫秒，所以，对比 `viewWillAppear:` 和 `viewDidLoad:`，`viewDidAppear:` 和 `viewWillAppear:` 之间的时间差可能会比较大。

  启动或恢复任何想要呈现给用户的视图动画。

- `viewWillDisappear:`

  该方法表示视图将要从屏幕上隐藏起来。这可能是因为其他视图控制器想要接管屏幕，或该视图控制器将要出栈。

  你可能会注意到，当此方法被调用时，没有办法能直接够判断这是由当前视图控制器要出栈还是其他视图控制器入栈导致的。

  区分的唯一方式是扫描当前视图控制器 `navigationController` 的 `viewControllers` 属性。为此，例 6-1 提供了一个代码框架。

  例 6-1　检测视图控制器的入栈和出栈

  ```
  -(void)viewWillDisappear:(BOOL)animated {
      NSInteger index = [self.navigationController.viewControllers
          indexOfObject:self];
      if(index == NSNotFound) {
          //即将出栈,销毁
      } else {
          //只是保存状态,暂停
      }
      [super viewWillDisappear:animated];
  }
  ```

  在使用 segue 和故事板时，展开的 segue 是更好的选择。苹果[1]的技术文档 TN2298 很好地概述了如何使用展开的 segue。

- `viewDidDisappear:`

  当上一个 / 下一个视图控制器的过渡动画完成时，此方法会被调用。正如 `viewDidAppear:`，`viewWillDisappear:` 事件也会有约 300 毫秒的差值。

注 1：iOS Developer Library，“Technical Note TN2298: Using Unwind Segues”(http://apple.co/1Hk6LP5)。

还记得图 6-1 中展示的生命周期图吗？再回过头看，你会发现这些生命周期方法有可能被调用多次，这取决于用户交互。在这些事件中还有可能出现无限循环的情况。

> 当应用处于后台或前台时，与视图可视性有关的生命周期方法不会被调用，而是通知 `UIAppDelegate`。只有应用处于前台时，视图控制器的生命周期方法才会被调用，且在过渡期不会被调用。
>
> 在视图控制器中注册 `UIApplicationDidBecomeActiveNotification`、`UIApplicationWillResignActiveNotification` 和 `UIApplicationWillEnterForegroundNotification` 通知来连接与视图可见性相关的回调。同时要记得在 `dealloc` 中注销。

以下列举了一些高效使用生命周期事件的最佳实践。

- 无需多说，不要重写 `loadView`。
- 将 `viewDidLoad` 作为最后的检查点，查看来自数据源的数据是否可用。如果可用，则更新 UI 元素。
- 如果每次都需要展示最新的信息，那么就使用 `viewWillAppear:` 更新 UI 元素。
  例如，在某一个消息应用中，如果在聊天会话中观看完共享视频后返回信息列表，那么用户必然希望看到最新的信息。

  但在新闻应用中，你可能不希望立即刷新列表，以免用户找不到上下文。在后一种情况下，列表视图控制器一般会监听来自数据源的事件，同时，应尽量精准、尽量少地更新文章列表。

- 在 `viewDidAppear:` 中开始动画。如果有视频等流式内容，那么就可以开始播放了。订阅应用事件来检测动画 / 视频或其他持续更新视频的处理是应该继续还是停止。
  不推荐在该方法中用最新的数据更新 UI。如果你这样做了，最终的效果是，在过渡动画完成之后，用户会过渡至旧的 UI，然后产生更新。这个体验不是很友好。

  话虽如此，但在一些使用案例中，你不得不在 `viewDidAppear:` 中执行 UI 更新。如果用户体验尚可接受，那就勉强这样吧。

- 使用 `viewWillDisappear:` 来暂停或停止动画。同样，不要做其他多余的操作。
- 使用 `viewDidDisappear`：销毁内存中的复杂数据结构。
  也可以在这里注销与视图控制器绑定的数据源通知，以及与动画、数据源、UI 更新有关的应用事件通知中心。

> 如果其他措施都不能优化载入时间，那么你可以在应用中增加一个精巧的动画。如此一来，你将会有数十秒的额外时间来完成任务，让用户在使用应用时不会有明显的延迟。需要注意的是，长时间的动画会惹恼用户，可能导致用户流失。这应该作为最后的方案，需要慎用。

6.2 视图

优化视图方面最具挑战性的部分是，很少有普适于所有视图的技术。每个视图都有其独特的用途，且大部分的优化技术都与特定的视图和暴露出的 API 有关。

在逐一讨论这些问题之前，我们先回顾以下的基本规则。

- 尽量减少在主线程中所做的工作。任何额外代码的执行都意味着更高的丢帧概率。过多的丢帧会导致不流畅。
- 避免较大的 nibs 或故事板。故事板很强大，但整个 XML 在真正使用之前必须被加载（I/O）和解析（XML 处理）。应该最小化故事板中的单元数目。
  如果需要的话，创建多个故事板或 nib 文件。这可以确保所有的屏幕并非都在应用启动时一次性加载，而是根据需要进行加载。这不仅有助于减少应用的启动时间，还能降低整体内存的消耗值。

  > 当 nib 文件被载入内存时，nib 加载代码需要执行几个步骤，以确保 nib 文件的对象被创建并正确地进行初始化。
  >
  > 当加载的 nib 文件中包含了对图片或声音资源的引用时，nib 加载代码会读取实际的图像或声音文件，将其放入内存并缓存。……在 iOS 中，只有图像资源存储在被命名的缓存中。[2]

- 避免在视图层次结构中多层嵌套。尽量保持扁平化。嵌套是必要的“罪恶”，但仍然是“罪恶”。
  在层次结构的任何位置添加视图时，它的祖先树节点会执行值为 `YES` 的 `setNeedsLayout:` 方法，当事件队列正在执行时，该设置会触发 `layoutSubviews:`。这个调用代价较大，因为视图必须根据约束重新计算子视图的位置，而且在祖先树的每一层都会发生这种情况。

  Twitter 团队贴出了一篇文章，他们放弃了包含 `UIImageView` 和一些 `UILabel` 的混合视图，转而创建了一个自定义视图，并对 `drawRect:` 方法进行了简单的优化。[3]

- 尽可能延迟加载视图并进行重用。更多的视图不仅会导致加载时间变长，还会使渲染时间变长，这些会影响内存和 CPU 的使用。
  如果需要，你可以创建自己的视图缓存。这可能会高出 `UITableView` 和 `UICollectionView` 已经提供的对单元格的复用支持。当视图不在视图窗口时，这些容器会释放视图。如果视图结构很复杂，并且比较耗费时间，那么实现自定义的视图缓存是个明智之举。

注 2：iOS Developer Library,“Resource Programming Guide: Nib Files”(https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/LoadingResources/CocoaNibs/CocoaNibs.html#//apple_ref/doc/uid/10000051i-CH4-SW9).

注 3：Twitter Blog,“Simple Strategies for Smooth Animation on the iPhone”(https://blog.twitter.com/2012/simple-strategies-for-smooth-animation-on-the-iphone).

如果使用 UIScrollView 呢？绝对是延迟加载。当滚动到位置 0 时才载入视图，然后通过建立自己的视图缓存，模仿 UITableView 的行为。将委托的 scrollViewDidScroll: 方法与 contentOffset（滚动位置）属性结合起来使用，以确定哪些视图需要渲染。

按照一般惯例，渲染的元素会超过视图窗口的屏幕高度，以避免滚动过程中出现抖动，因为滚动开始时，这些元素需要迅速地被渲染出来。

需要牢记的是，UITableView 继承自 UIScrollView，这意味着，如果 UITableView 可以做智能视图缓存，那自定义代码也可以实现。

- 对于复杂的 UI 而言，最好使用自定义绘图。这样只会触发一个视图进行绘制，而不是多个子视图，同时也避免了调用代价较高的 layoutSubviews 和 drawRect: 方法。
此外，要避免使用具有通用目的及功能丰富的组件而带来的消耗，你可以使用那些直接实现了绘制方法的视图来代替。

 例如，如果想要显示纯文本，你不需要使用繁重的 UILabel（见图 6-4）。

介绍完这些基本规则后，接下来我们将探讨一些更常见的视图，并深入了解与每种视图相关的性能技巧。

6.2.1 UILabel

这可能是 iOS 上最常用的视图了。它虽然看起来简单，但是渲染代价却不容小觑。下列是涉及的一些复杂步骤。

(1) 使用字体、字体类型以及要被渲染的文本时，计算需要的像素数目。这是一个消耗较大的过程，应尽可能少地去做。[4]
(2) 检查要被渲染的宽度。
(3) 检查 numberOfLines，计算将要展示的行数。
(4) sizeToFit 是否被调用？如果是，则计算高度。
(5) 如果 sizeToFit 没有被调用，检查当前的内容能否在给定的高度下展示出来。
(6) 如果 frame 不够，使用 lineBreakMode 确定隐藏或截断的位置。
(7) 注意其他的配置选项，如图 6-4 所示（例如，是纯文本还是属性文本、阴影、对齐、自动收缩等）。
(8) 最后，使用字体、类型及颜色来渲染最终展示的文本。

具体说明每个 UILabel 是一件工作量很大的事情。使用较少的标签，更容易管理效果，使用较多的标签，你就需要多留意这些标签的创建、配置和重用。

注 4：所需的尺寸计算必须在主线程中完成。

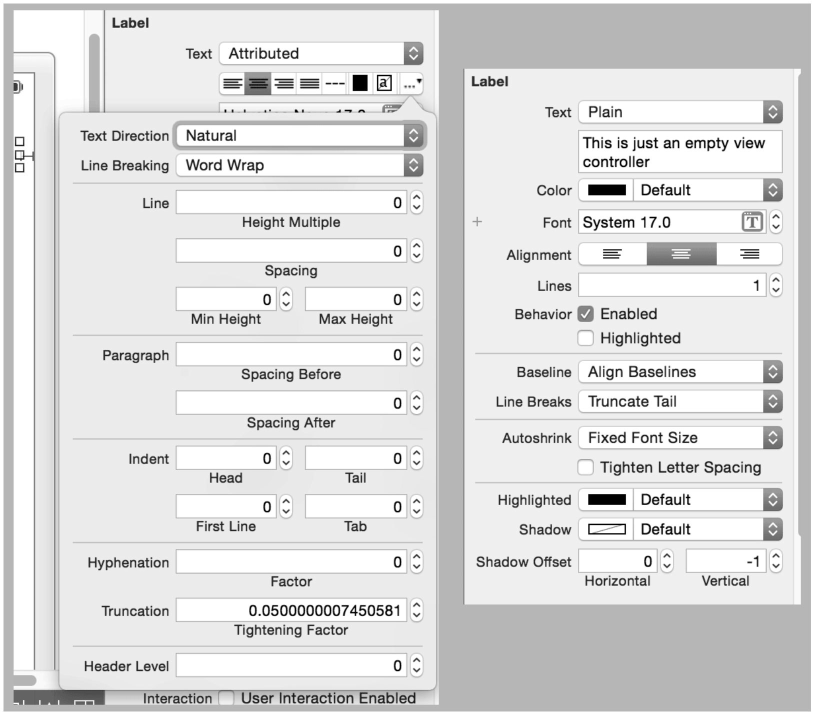


图 6-4：`UILabel` 选项

如果将动态计算出的标签宽度作为容器宽度的一小部分，那么你要确保宽度可以由某一百分比均匀分配。例如，两个标签各占容器宽度的 50%，那么容器宽度必须是偶数。避免 `width/2` 这样的调用。如果宽度很小，其他的都不会受影响，除了渲染，因为渲染需要反锯齿，这是一个代价很大的操作。

6.2.2 UIButton

按钮几乎无处不在，如导航控制器中的导航按钮、消息应用中的“发送”按钮、自定义表单中的“发送”按钮，等等。所以，除非应用只有动画和自定义渲染，否则应用中总会有一个按钮。

渲染按钮的方式有以下四种：

- 使用自定义文本的默认渲染
- 全尺寸资源的按钮
- 可变大小的资源
- 使用 CALayer 和贝塞尔路径自定义绘制

我们不会探究每一个细节，主要关注每一个选项的优点和缺点。第一个选项是非常简单的，其余的都在 Designing for iOS: Taming UIButton（https://robots.thoughtbot.com/designing-for-

ios-taming-uibutton）中进行讨论。

表 6-1 列出了使用每个选项渲染按钮的利与弊。

表6-1：按钮渲染选项

选项	优点	缺点
自定义文本	最简单的方式，可直接使用	通常是比较呆板、毫无装饰的按钮
全尺寸资源	可自定义的背景 无需代码即可实现 可实现 A/B 测试——图像可在运行实验时下载	图片打包在应用中，导致包变大
可变大小资源	可自定义的背景 无需代码即可实现 可实现 A/B 测试——图像可在运行实验时下载 包大小的增量相对较小	资源的任何更改可能都需要重新计算 / 重置 `UIEdgeInsets` 值
使用 CALayer 和贝塞尔路径自定义绘制	完全是自定义绘图	任何格式的更改或升级都可能需要更新应用

你需要权衡这些选项的利弊，并选择适合自己需要的那个。按钮是一个可以被渲染的简单组件，几乎不需要其他性能辅助。但如果想要使其更加美观、靓丽和精致，你还需要探寻更多其他的可用选项。

6.2.3 UIImageView

每个应用都含有图像，因为图像让应用看起来更加漂亮。

但在渲染代价较大的各种 UI 元素中，图像首屈一指。它们大多都是固定的，一旦创建就不会改变。要想显示图像的变化，就得加载另外一个图像。iOS 仍旧不支持 GIF 动画。你只能创建 `animationImages` 的一个数组来存放可以生成动画的图片。一些其他的选择包括了使用自定义编码或使用第三方库，如 ImageMagick（http://www.imagemagick.org/）或 `AnimatedGIFImageSerialization`（https://github.com/mattt/AnimatedGIFImageSerialization）。

在使用 `UIImage` 和 `UIImageView` 时，遵循以下的最佳实践可以提升性能。

- 对于已知的图像，使用 `imageNamed:` 方法加载图像。它可以确保内容只被加载至内存一次，还可以确保在多个 `UIImage` 对象间改变用途。
- 在使用 `imageNamed:` 方法加载包图片时，使用资源包。如果应用有一堆图标，且每个图标都较小时，这种方式极其有用。可以随意地创建相关图像（即通常被一起使用的图片）的多个目录。

  当 iOS 从磁盘加载时，有一个最佳的缓冲区大小可以用来在单独的读操作中加载多个图片。此外，与开启一个流并从这个流中读取多张图片相比，开启多个 I/O 流会有一些其他支出。一般情况下，读取一个 32KB 的组合文件比读取每个 2KB、共 16 个文件更快。

  但是，如果你想加载一个只使用一次的大图像，最好谨慎思考一下，考虑使用 `imageWithContentsOfFile:`（https://developer.apple.com/library/ios/documentation/

UIKit/Reference/UIImage_Class/index.html#//apple_ref/occ/clm/UIImage/imageWithContentsOfFile:）代替资源目录和 imageNamed: 方法，因为资源目录缓存了这些图片（在这个情况下并不需要进行缓存）。

在我之前开发的一款应用中，团队成员发现，在一个包中的资源目录初始化两屏所需的图片，可以将初始加载时间减少大约 300 毫秒。

- 对于其他图像，使用高性能的图像缓存库。AFNetworking 和 SDWebImage 都是可选的强大库。
  当使用内存中的图片时，确保正确配置了内存的使用参数。不要使用硬编码。让它能够自适应——使用合理的 RAM 百分比可以较好地进行配置。
- 载入的图像与即将渲染的 UIImageView 大小相同。如果被解析的图像尺寸与 UIImageView 相同，那么你会得到极高的性能，因为调整图像大小是一个耗费较大的操作，如果该图像被包含在 UIScrollView 中，则耗费会更大。
  如果图像来自网络下载，那么尽量下载和视图大小匹配的图像。如果行不通，适当地对图片进行预处理，调整其大小。
- 如果需要使用一些类似于模糊或色调的效果，那么可以创建一份图像内容的副本，在副本上施加效果，然后使用最终的位图创建所需的 UIImage。如此一来，这些附加的效果只会被使用一次，如果有需要，原始图像还可以用于其他显示。
- 无论使用何种技术加载图像，在非主线程中执行，最好在一个专用的队列中执行。
  尤其要在非主线程中解压 JPG/PNG 图像。
- 最后同样重要的是，确定是否真的需要图像。如果要展示一个评分栏（http://stackoverflow.com/questions/27600288/rating-bar-like-android-in-codename-one），最好使用直接绘制的自定义视图，而不是使用多个图像，通过调整透明或覆盖来实现。

6.2.4 UITableView

无论是在新闻应用、邮件应用、照片流，还是其他的应用中，UITableView 都是最常用于显示数据的视图。UITableView 提供了一个展示信息条的极好选择，这些信息条既可以是同一类别，也可以是不同类别。

UITableView 绑定了两个协议。

- UITableViewDataSource
  必须将 dataSource 属性设置到数据源上。顾名思义，数据源是指将要填充至列表单元格中的数据源。
- UITableViewDelegate
  必须将 delegate 属性设置到委托上，当用户与列表或单元格交互时，此处的委托必须能接收到回调。

这些协议和 UITableView 之间的部分逻辑关系如图 6-5 所示。

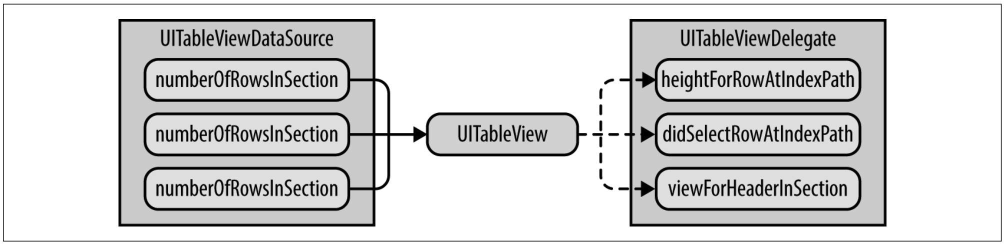


图 6-5：UITableView、UITableViewDataSource 和 UITableViewDelegate

下列是使用 UITableView 时需要牢记的一些最佳实践。

- 在数据源的 tableView:cellForRowAtIndexPath: 方法中，使用 tableView:dequeueReusableCellWithIdentifier: 或 tableView:dequeueReusableCellWithIdentifier:forIndexPath: 进行单元格的重用，而不是每次都创建新的单元格。
  单元格的创建有性能成本。如果几个单元格必须在很短的时间间隔内被创建（例如，当用户滚动列表视图时），那么这个成本会成倍增长。同时，单元格超出范围后会被释放，这将造成双倍打击。重用单元格意味着只存在渲染单元格这一种开销。
- 尽可能避免动态高度的单元格。诚然，已经确定的高度代表着只需很少的计算量。如果内容是动态配置的，那么不仅需要计算高度，而且每次视图要被渲染时，单元格的内容也需要刷新和重新布局。这是一个很大的性能损失。
  图 6-6 展示了填充固定高度和可变高度的 UITableView 的例子。如果你想要使用可变高度的单元格，最好选择较高的单元格，因为这只需要为较少的单元格计算高度，从而减少了计算量。

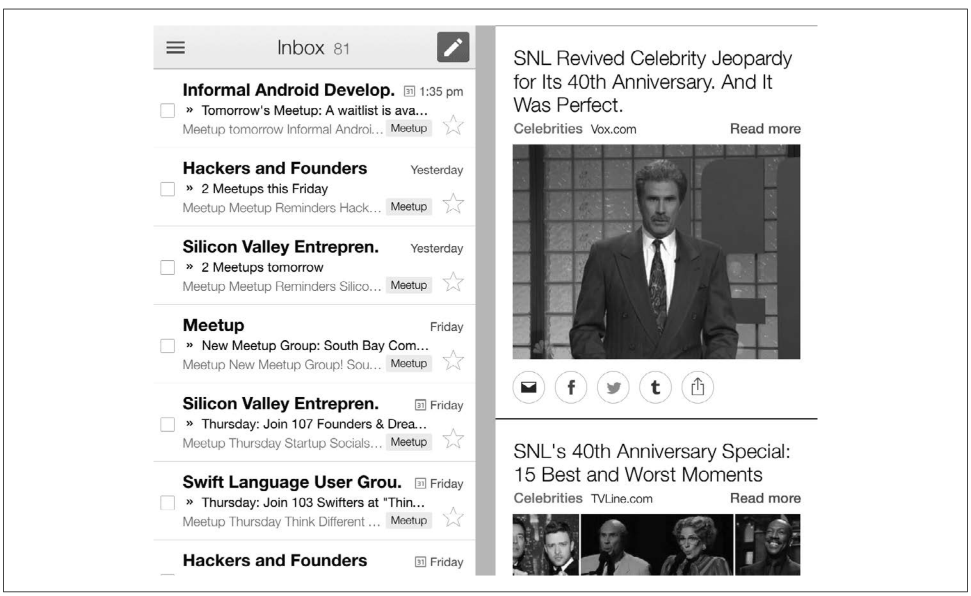


图 6-6：UITableView（左侧是固定高度的单元格，右侧是可变高度的单元格）

- 如果你真的需要动态高度的单元格，那么定义一个规则来标记单元格是脏的。如果某个单元格是脏的，计算它的高度并缓存。在委托的 `tableView:heightForRowAtIndexPath:` 回调中继续返回缓存的高度，直到单元格不再被标记为脏。
  如果要被渲染的模型是不可变的，一个可用的简单规则是，检查当前被渲染的模型是否和相应的 `indexPath` 的值一样。如果一样，则使用同样的值渲染，无需进一步的处理。如果不一样，则重新计算值，并将新的对象（模型）附加至该单元格。
- 当用自定义视图重用单元格时，要避免通过调用 `layoutIfNeeded` 每次都对其进行布局。即使一个单元格的高度是固定的，也有可能出现这样的情况：在单元格中的独立元素可能会被设置成不同的高度，例如，`UILabel` 支持多行内容，`UIImageView` 可以装入不同大小的图像。

  需要避免这种情况。固定每个元素的尺寸。这可以确保单元渲染所需的时间达到最小值。
- 避免透明的单元格子视图。创建 `UITableViewCell` 时，尽量引入不透明元素。半透明或透明元素（`alpha` 低于 1.0 的视图）很好看，但会有性能损失。
  出于美学考虑，你可能仍然希望将 `alpha` 设置为小于 1.0 的值。那么你就需要注意成本了。
- 在快速滚动时考虑使用界面外壳（见图 6-7）。当用户快速滚动列表视图时，虽然使用了所有的优化，但视图的重用和渲染仍然需要超过 16 毫秒，还有可能出现偶发的丢帧现象，从而导致不流畅的体验。



图 6-7：使用有外壳的界面

在这些情况下，使用一个界面外壳是一个较好的选择，外壳可以被预先定义，它的唯一目的就是告诉终端用户这些部分即将展示一些数据。当滚动速度降低，并低于阈值时，刷新最终的视图并填充数据。

你可以使用与列表视图相关联的 `panGestureRecognizer` 属性获取速度值（见例 6-2）。

例 6-2　列表视图的速度

```
-(void)scrollViewDidScroll:(UIScrollView *)scrollView {
    CGPoint velocity = [tableView.panGestureRecognizer
                         velocityInView:self.view];
    self.velocity = velocity;
}

-(UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    if(fabs(self.velocity.y) > 2000) {
        //返回界面外壳
    } else {
        //返回真正的单元格
    }
}
```

- 避免渐变、图像缩放以及任何屏幕外的绘制。这些效果对 CPU 以及图形处理单元（GPU）来说都是消耗。

6.2.5　UIWebView

`UIWebView` 是用于渲染*未知或动态*内容的最常见视图。通常情况下，你会将 web 视图指向一些内嵌的 HTML 或 web URL。

虽然有些应用可能全部都是原生的，但还是有需要使用 `UIWebView` 的场景，以下是一些常见场景。

- 任何应用中的用户登录。Spotify、Mint 和 LinkedIn 这样的应用使用原生 UI 渲染登录表单。但这有一定的限制。
  例如，如果想使用 CAPTCHA 筛选刷屏的机器人，你就需要为所有的格式（http://www.theverge.com/2014/12/3/7325925/google-is-killing-captcha-as-we-know-it）提供支持，并将其打包至应用中，或将用户指向一个网页登陆 URL，让服务器生成任何需要的复杂 UI。
- 在任何应用中显示隐私政策或使用条款。因为这些会随着时间变化，并且需要大量的格式化（文本样式、编号列表、其他内容的交叉引用），使用原生视图不是较好的选择。
- 新闻或文章阅读器，因为大部分的文章都是为 Web 创建的，几乎都是 HTML。
- 邮件应用。例如，初始邮件是 HTML 形式，当呈现消息或跟帖，以及撰写回复时。

> 如果你需要展示较小的富文本，使用 `UILabel` 的 `NSAttributedString`。无需 CSS 或 JavaScript 的支持。它是一个有关联属性（如字体和字间距）集合的字符串，这些属性可以用于字符串中单独的字符或某一范围内的字符。

使用 UIWebView 时，请将以下几个最佳实践牢记在心。（需要注意的是，关于 UIWebView 能做的事情非常少，并非都是关注性能的；相反，此处的重点是以最恰当的方式展示 HTML 内容。）

- UIWebView 可能比较笨重且迟钝，所以尽可能复用 web view。同时，UIWebView 也因内存泄漏而知名。因此，每个应用的实例都应该足够好。
  无论何时想向用户展示新的 URL，先将内容重置为空的 HTML。这样就能确保 web view 不会将之前的内容展示给用户。要想实现这一功能，在 loadRequest: 方法后调用 loadHTMLString:baseURL: 即可。
- 附加一个自定义的 UIWebViewDelegate。实现 webView:shouldStartLoad WithRequest: navigationType: 方法。要留意 URL scheme。如果是 http 或 https 以外的东西，需要注意：应用应该知道如何处理这种情况，或警告用户该网站正试图脱离应用。
  这是一个较好的做法，不仅能保证用户不会突然出现在另一个应用当中，同时也对恶意内容进行了防护，尤其是恰巧要展示一个未知 URL 的内容时——例如，在邮件或消息应用中。
- 你可以通过 stringByEvaluatingJavaScriptFromString: 方法创建一个桥来连接应用和 JavaScript，从而在当前已经加载的 web 页面执行 JavaScript。如果想要调用原生应用的方法，你可以参考之前的处理方法，使用自定义的 URL scheme。
- 实现委托的 webView:didFailLoadWithError: 方法，以保持对所有可能出现的错误的紧密追踪。
- 实现 webView:didFailLoadWithError: 方法来处理特定的错误，如例 6-3 所示。如果域名与 NSURLErrorDomain 相等，那么 NSError 对象是有不同意义的。

例 6-3　用 UIWebView 处理错误

```
-(void)webView:(UIWebView *)webView
    didFailLoadWithError:(NSError *)error {
    if([NSURLErrorDomain isEqualToString:error.domain]) {
        switch(error.code) {
            case NSURLErrorBadURL:
                //处理错误的URL
                break;
            case NSURLErrorTimedOut:
                //处理超时
                break;
            //……等等
        }
    }
}
```

- UIWebView 不会通知任何的 HTTP 协议错误，例如响应是 404 或 500 的错误。如例 6-4 所示，你需要触发两次调用，第一次使用自定义的 NSURLConnection 调用，然后是通过 web view 的调用。你可以提供一个 NSURLConnection 的委托，然后实现 connection:didReceiveResponse: 方法，以便获取响应的相关信息。

例 6-4　UIWebView 和 HTTP 错误

```
@interface HPWebViewController() <UIWebViewDelegate,
```

```
                                        NSURLConnectionDataDelegate>
@property （nonatomic，assign） BOOL shouldValidate;

@end

@implementation HPWebViewController

-(BOOL)webView:(UIWebView *)webView
    shouldStartLoadWithRequest:(NSURLRequest *)request
    navigationType:(UIWebViewNavigationType) navigationType {

    if(self.shouldValidate) {
        [NSURLConnection connectionWithRequest:request delegate:self];
        return NO;
    }

    return YES;
}

- (void)connection:(NSURLConnection *)connection
    didReceiveResponse:(NSURLResponse *)response {

    NSInteger status = [(NSHTTPURLResponse *)response statusCode];
    if(status >= 400) {
        //哇！一个错误
        //展示警报或隐藏web view——不要展示错误的网页
    } else {
        self.shouldValidate = NO;
        [self.webView loadRequest:connection.originalRequest];
    }
    [connection cancel];
}

@end
```

因为这种技术需要加载网页两次，所以是不受推荐的。当加载页面时，网页视图是可以展示错误的。也许就是点击了接收的某个消息中的一个连接，用户才发起了请求。

- 嵌入了 `UIWebView` 的容器应该提供以下元素。
  - 导航按钮（后退和前进）。
  - 重载按钮。
  - 取消按钮，用于取消当前正在加载的页面。
  - 用于展示页面标题的 `UILabel`。
  - 用于退出 web view 的关闭按钮。如果应用（如混合应用）只有这一个唯一的界面，则不需要关闭按钮。

> 混合应用是通过 UIKit 嵌入的 HTML 应用，特别是 `UIWebView` 或新的 `WKWebView`。本书不讨论混合应用的性能（这个话题本身就可以写一本书）。

> **iOS 8 的新特性：WebKit**
>
> iOS 8 的 WebKit 特性（https://developer.apple.com/library/ios/documentation/Cocoa/Reference/WebKit/ObjC_classic/index.html#//apple_ref/doc/uid/TP30000745）比 `UIWebView` 的性能更好。如果你正在写一个新的应用，最好使用 `WKWebView`。但请记住，如果你选择使用 `WKWebView`，在 iOS 7 设备上需要回退到 `UIWebView`。
>
> 使用 `WKWebView` 的基本规则和之前讨论的 `UIWebView` 的规则是相同的。作为附注，你可以下载一个应用（https://itunes.apple.com/app/id928647773?mt=8）来测试这两者之间的区别。[5]

6.2.6 自定义视图

在非游戏应用中或不以动画为核心的应用中，从头开始写自定义视图并不常见。比较常用的方法是，使用 Interface Builder 和自定义 nib 文件创建复合视图。

虽然这是一个极好的初级技巧，但是，一旦创建了更复杂的 UI，或在列表视图中使用了复合视图，那性能下降的问题就会暴露出来。

Twitter 团队在其应用开发的早期遇到了这个问题，他们摒弃了复合视图的使用，改为直接绘制视图，从而最大程度地减少了渲染视图所需的消耗。[6]

图 6-8 展示了一条推文的 UI。



图 6-8：渲染的推文

UI 的基本实现可能包含以下内容（见图 6-9）。

(1) `UIImageView` 作为头像图片。
(2) 用 `UILabel` 的 `NSAttributedText` 展示用户名称。
(3) 推文的内容使用 `detectorType = UIDataDetectorTypeLink` 的 `UITextView`，因为可能包含链接。
(4) 数据使用 `UILabel`。

注 5：免责声明：我并不认可该应用。

注 6：Twitter Blog,“Simple Strategies for Smooth Animation on the iPhone”(https://blog.twitter.com/2012/simple-strategies-for-smooth-animation-on-the-iphone).



图 6-9：原生推文的视图

每个视图使用额外的合成负责核心动画。此外，每当子视图改变时，创建嵌套的层次会导致 `layoutSubviews` 被调用多次。

为了优化性能，通过在一个 `drawRect:` 中绘制全部的元素，该团队创建了一个自定义视图，如图 6-10 所示。



图 6-10：自定义推文视图——直接绘制

假设有一个邮件应用，我们需要在收件箱中显示邮件的概览，概览应包含以下细节：

- 发送者的姓名 / 邮件 ID
- 发送的日期或时间
- 主题
- 内容的一些片段（少数主要内容）
- 提示邮件是新的、已经被阅读的，还是已经回复了的标记
- 选择多封邮件的选择器（也可能只是一个复选框）
- 邮件是否有附件的标记

最终的布局应该与图 6-11 类似。

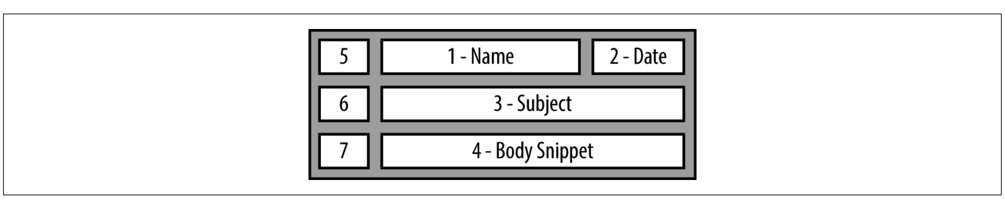


图 6-11：邮件概览

下面将以两种样式（复合视图和自定义视图）创建该视图，并测量这两种视图的性能（创建一个单元格的时间和渲染所需的时间）。我们还将分析使用每种样式的其他优缺点。

1. 复合视图

创建复合视图很简单。你只需要创建一个继承自 `UITableViewCell` 的新的视图类即可。

以下是你需要遵循的一些步骤。

(1) 导航到 File→New→File。
(2) 选择 iOS→Source→Cocoa Custom Touch。
(3) 类名是 `HPMailCompositeCell`，它是 `UITableViewCell` 的子类。
(4) 勾选 Also create XIB file 的选项。
(5) 点击完成。

添加四个 `UILabel`、两个 `UIImage` 和一个 `UIButton` 元素，对它们进行排列，使得最终的结构和图 6-12 中的相匹配。

> 针对复杂视图，在动画过程中使用视图光栅化，包括但不限于滚动。
> 通常情况下，滚动期间的视图布局是不改变的，在动画过程中，将 `UIView` 的 `layer` 的 `shouldRasterize` 属性设置为 `YES`，在动画完成之后，将其设置为 `NO`。[7]

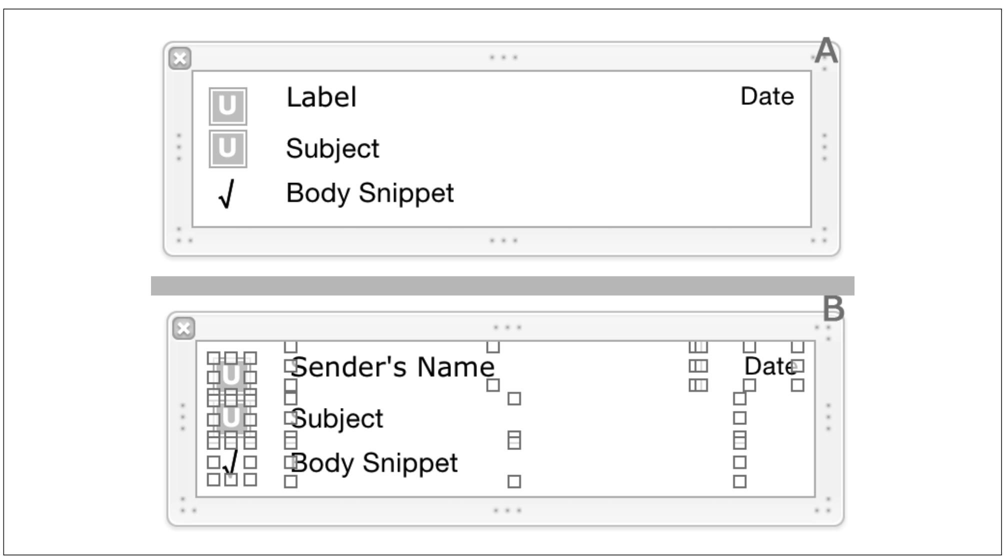


图 6-12 复合视图——布局（A）和独立（B）视图

2. 直接绘制

为了创建直接绘制的自定义视图，我们再次创建一个继承自 `UITableViewCell` 的新类，但

注 7：Stack Overflow,“When Should I Set layer.shouldRasterize to YES?”(http://stackoverow.com/a/19408290).

是不勾选创建一个 XIB 文件的选项。

我们将这个类命名为 HPMailDirectDrawCell。

像之前讨论的那样，我们需要属性来支持细节。最后，覆盖 drawRect: 方法以自定义渲染元素。

例 6-5 展示了渲染所有元素的代表性代码。

例 6-5　直接绘制的自定义视图

```
//HPMailDirectDrawCell.h
typedef NS_ENUM(NSInteger, HPMailDirectDrawCellStatus) {
    HPMailDirectDrawCellStatusUnread,
        HPMailDirectDrawCellStatusRead,
        HPMailDirectDrawCellStatusReplied
};

@interface HPMailDirectDrawCell : UITableViewCell

@property (nonatomic, copy) NSString *email;
@property (nonatomic, copy) NSString *subject;
@property (nonatomic, copy) NSString *date;
@property (nonatomic, copy) NSString *snippet;
@property (nonatomic, assign) HPMailDirectDrawCellStatus mailStatus;
@property (nonatomic, assign) BOOL hasAttachment;
@property (nonatomic, assign) BOOL isMailSelected;

@end

@implementation HPMailDirectDrawCell
//覆盖所有的初始化器—出于简洁省略了

//覆盖drawRect方法
-(void)drawRect:(CGRect)rect {
    {
                UIImage *statusImage = nil;
                switch(self.mailStatus) {
                        case HPMailDirectDrawCellStatusRead:
                                statusImage = [UIImage imageNamed:@"mail_read"];
                                break;
                        case HPMailDirectDrawCellStatusReplied:
                                statusImage = [UIImage imageNamed:@"mail_replied"];
                                break;
                        case HPMailDirectDrawCellStatusUnread:
                        default:
                                statusImage = [UIImage imageNamed:@"mail_new"];
                                break;
                }

                CGRect statusRect = CGRectMake(8, 4, 12, 12);
                [statusImage drawInRect:statusRect];
        }

        {
```

```
            UIImage *attachmentImage = nil;
            if(self.hasAttachment) {
                    attachmentImage = [UIImage imageNamed:@"mail_attachment"];
        CGRect attachmentRect = CGRectMake(8, 20, 12, 12);
       [attachmentImage drawInRect:attachmentRect];
            }
       }

    {
            UIImage *selectedImage = [UIImage imageNamed:
          (self.isMailSelected ? @"mail_selected": @"mail_unselected")];
      CGRect selectedRect = CGRectMake(8, 36, 12, 12);
    [selectedImage drawInRect:selectedRect];

      //或者,能够使用Core Graphics绘制矢量图像
      }

      CGFloat fontSize = 13;
      CGFloat width = rect.size.width;
      CGFloat remainderWidth = width - 28;
      {
              CGFloat emailWidth = remainderWidth - 72;
              UIFont *emailFont=[UIFont boldSystemFontOfSize:fontSize];
              NSDictionary *attrs = @{ NSFontAttributeName: emailFont };

              [self.email drawInRect:CGRectMake(28, 4, emailWidth, 16)
          withAttributes:attrs];
      }

      {
              UIFont *stdFont = [UIFont systemFontOfSize:fontSize];
              NSDictionary *attrs = @{ NSFontAttributeName: stdFont };
              [self.subject drawInRect:CGRectMake(28, 24, remainderWidth, 16)
          withAttributes:attrs];
              [self.snippet drawInRect:CGRectMake(28, 44, remainderWidth, 16)
          withAttributes:attrs];
      }

      {
              UIFont *verdana = [UIFont fontWithName:@"Verdana" size:10];
              NSDictionary *attrs = @{ NSFontAttributeName: verdana };
              [self.date drawInRect:CGRectMake(width - 60, 4, 60, 16)
          withAttributes:attrs];
      }
}
@end
```

现在有两个需要进行比较的方面：运行时性能和代码维护。

与预计的一样，直接绘制的自定义视图中的运行时性能更好。差异是什么呢？我们来看看图 6-13 中的数字。

```
0.177 HPerf Apps[3307:891359] [I] [prepareForSegue] i=ch_08_10_cpsv, row=3
0.226 HPerf Apps[3307:891359] [I] [cell 0]: Time=17675125 ns
0.232 HPerf Apps[3307:891359] [I] [cell 1]: Time=1980958 ns
0.237 HPerf Apps[3307:891359] [I] [cell 2]: Time=1913166 ns
0.241 HPerf Apps[3307:891359] [I] [cell 3]: Time=1937208 ns
0.245 HPerf Apps[3307:891359] [I] [cell 4]: Time=1838375 ns
0.249 HPerf Apps[3307:891359] [I] [cell 5]: Time=1976000 ns
0.253 HPerf Apps[3307:891359] [I] [cell 6]: Time=1944208 ns
0.257 HPerf Apps[3307:891359] [I] [cell 7]: Time=1852250 ns
0.785 HPerf Apps[3307:891359] <HPInst> SCR_Views_Custom_Composite
3.290 HPerf Apps[3307:891359] [I] [cell 8]: Time=6602208 ns
3.341 HPerf Apps[3307:891359] [I] [cell 9]: Time=9596000 ns
3.352 HPerf Apps[3307:891359] [I] [cell 10]: Time=130708 ns
3.380 HPerf Apps[3307:891359] [I] [cell 11]: Time=79916 ns
4.383 HPerf Apps[3307:891359] [I] [prepareForSegue] i=ch_08_10_cpsv, row=4
4.397 HPerf Apps[3307:891359] [I] [cell 0]: Time=359041 ns
4.398 HPerf Apps[3307:891359] [I] [cell 1]: Time=177416 ns
4.400 HPerf Apps[3307:891359] [I] [cell 2]: Time=199041 ns
4.401 HPerf Apps[3307:891359] [I] [cell 3]: Time=125625 ns
4.401 HPerf Apps[3307:891359] [I] [cell 4]: Time=110541 ns
4.403 HPerf Apps[3307:891359] [I] [cell 5]: Time=215166 ns
4.403 HPerf Apps[3307:891359] [I] [cell 6]: Time=120583 ns
4.405 HPerf Apps[3307:891359] [I] [cell 7]: Time=222458 ns
4.924 HPerf Apps[3307:891359] <HPInst> SCR_Views_Custom_Composite
5.649 HPerf Apps[3307:891359] [I] [cell 8]: Time=293375 ns
5.680 HPerf Apps[3307:891359] [I] [cell 9]: Time=133250 ns
5.713 HPerf Apps[3307:891359] [I] [cell 10]: Time=59208 ns
5.730 HPerf Apps[3307:891359] [I] [cell 11]: Time=28083 ns
```

图 6-13：复合视图对比直接绘制——比较初始化和重用

根据例 6-5 中给出的代码，表 6-2 总结了复合视图和直接绘制的任务时间。

表6-2：数字比较

任务	复合视图	直接绘制
首次初始化	17.6 毫秒	0.36 毫秒
后续初始化	1.8~1.9 毫秒	0.1~0.2 毫秒
滚动后的首次初始化	6.6 毫秒	0.3 毫秒
滚动后的第二次初始化	9.6 毫秒	0.13 毫秒
重用	0.08~0.13 毫秒	0.03~0.08 毫秒

你可能会注意到，性能有 2~20 倍的惊人差异。使用直接绘制时，初始加载速度会快 50 倍。

而且这些数据还是在直接绘图的代码没有优化的情况下得出的。

因此，从性能角度来看，在某些时候，直接绘图提供的性能比复合视图提供的要好一个数量级。

然而，从维护的角度来看，代码会难以维护和发展。一旦应用稳定下来，你就可以比较明确地将复合 UI 换成直接绘图。

此外，如果创建的视图仅包括标准控件，做 A/B 测试时，比较容易的方式是根据不同的设备发送新的 nib 文件，然后从 nib 文件加载 UI。这样的话，你可以展示不同的布局，而无需发行应用的新版本。

6.3 自动布局

iOS 6 推出了自动布局，减轻了在复杂屏幕中调整元素的痛苦。通过被称为约束的东西，自动布局可以描述视图相互之间的位置、容器、大小等。约束可以描述一个元素距另一元素的距离（水平或垂直）、其大小（宽度或高度），或其与另一元素的对齐方式（水平或垂直）。

这里假设你已经具备了自动布局的知识。如果没有，你应该回顾 iOS 开发者库的官方参考（https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/AutolayoutPG/）或 Matthijs Hollemans 的 Beginning Auto Layout Tutorial in iOS 7: Part 1（http://www.raywenderlich.com/50317/beginning-auto-layout-tutorial-in-ios-7-part-1）。

虽然自动布局是一个不错的方法，它允许你将元素的位置、大小交给核心引擎，而不是在代码中实现，但这却带来了性能开销。自动布局的实现涉及了求解线性方程组，这些方程组就是为了满足约束而列出的。它使用 Cassowary（http://stacks.11craft.com/cassowary-cocoa-autolayout-and-enaml-constraints.html）约束求解工具包。作为任何通用方程的解答器，它的复杂度为 O（*N*），其中 *N* 是约束的数目，而不是元素的数量。这意味着，在一般情况下，为了确定视图中所有元素的位置和大小，可能有大概 4*N* 个方程要求解，而且解方程花费的时间与元素的个数和包含的约束个数是不成比例的。

在 Florian Kugler 的实验中，[8] 如果视图的数量增加至几百个，自动布局需要几十秒或更多的时间，而直接设置结构大小在毫秒级别就可完成。一般情况下，直接设置结构大小要比使用自动布局快 1000 倍左右。Kugler 的测试结果如图 6-14 所示。你可以在 Github 上找到用来测试的应用源代码（https://github.com/oriankugler/AutoLayoutProling）。

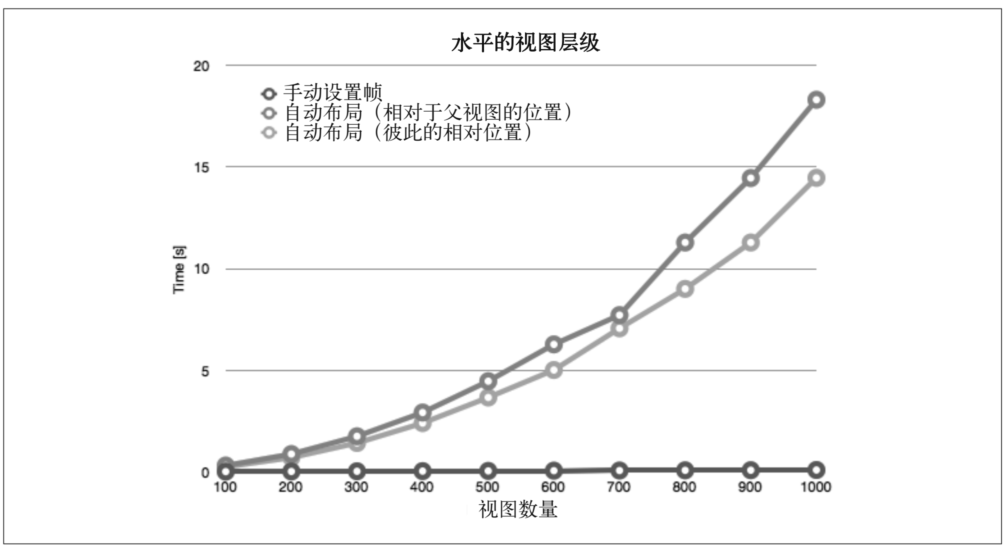


图 6-14：自动布局性能

注 8：Florian Kugler, “Auto Layout Performance on iOS” (http://floriankugler.com/2013/04/22/auto-layout-performance-on-ios/).

比较有趣的是，自动布局使用本地约束（例如，元素彼此之间的位置关系）会比使用全局约束更快，其中全局约束指相对于父类视图的位置。

任何自定义代码将会对视图有专门的了解，如此一来，与使用通用的方程解法来确定视图的位置和大小相比，这种渲染的速度更快。

话虽如此，当 Kugler 对更多真实的应用进行测量时，测量数字表明，虽然自动布局较慢，但在测试应用中慢的程度没有超过 10 倍或 100 倍。例如，在图 6-15 中所展示的应用中，自动布局花费了大约 180 毫秒，而自定义代码花费了大约 120 毫秒。所以，虽然自动布局需要 50% 额外的时间，但它仍然不是一个十分糟糕的选择。



图 6-15：测试自动布局的真实应用

对自动布局的最终裁决是什么？靠自己的判断。衡量展示视图和渲染视图所需的时间。如果超过了阈值，考虑使用自定义代码。阈值根据具体的应用而定。请记住，使用自动布局时，布局性能和渲染性能总会有提高的空间。

就自定义代码而言，每次布局更新时，总有搬运和测试代码的负担。这也意味着，你可能需要放弃使用自定义模板运行 A/B 测试的方案。

6.4 尺寸类别

直到 iPhone 4S，应用的开发都是简单的，因为只需要为一种大小和分辨率进行开发。iPhone 5 和 5S 的垂直尺寸都有所增加。iPhone 6 和 6 Plus 在水平和垂直方向都增加了更多的像素。在 iPad 系列中，iPad 4 在上一代的基础上将分辨率增加了一倍。每英寸的像素（pixels per inch，ppi）增长了 3 倍，因此，应用的设计人员和开发人员需要将两倍的图像进行打包（这使得包的下载更难），更别提直接绘图时的庞大像素数量了。

表6-3：iOS设备——屏幕分辨率和密度

设备	屏幕分辨率	像素密度 (ppi)
iPhone 3G	320 × 480	163
iPhone 4	640 × 960	326
iPhone 4S	640 × 960	326
iPhone 5	640 × 1136	326
iPhone 5S	640 × 1136	326
iPhone 6	750 × 1334	326
iPhone 6 Plus	1080 × 1920[9]	401
iPad 2	1024 × 768	132
iPad（第三代）	2048 × 1536	264
iPad（第四代）	2048 × 1536	264
iPad Air	2048 × 1536	264
iPad Air 2	2048 × 1536	264
iPad Mini	1024 × 768	163
iPad Mini (Retina)	2048 × 1536	325

iOS 渲染的特点是点的概念，这是与密度无关的分辨率。注意，ppi 不是点到像素的比例。将点当作 iOS 提供的一个缩放因素，那么你就不用担心缩放本身了。因此，一个 10pt 视图可能对应 iPhone 3G 的 10 像素、iPhone 4 和 5S 的 20 像素，以及 iPhone 6 Plus 上的 30 像素。定义约束时是以点为单位的，如图 6-16 所示。字体大小同样以点为单位。

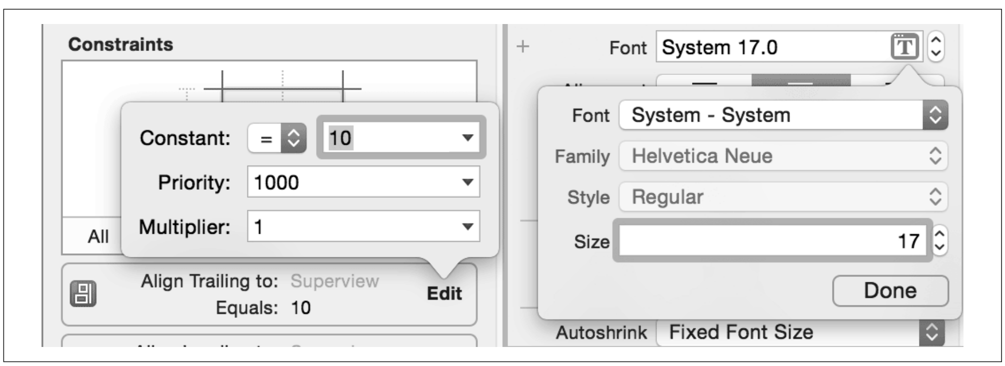


图 6-16：视图约束

当使用直接绘图的方式创建自定义视图时，`drawRect:` 方法给出了大规模的 `CGRect` 尺寸。针对之前创建的自定义单元格，你需要注意：在 iPhone 5S 和 iPhone 6 上展示的大小是 320x64。

苹果公司没有向开发者暴露确切的像素尺寸，取而代之地做了一件非常出色的事情，即通过所谓的*尺寸类别*提取出配置。尺寸类别标识了要展示的高度和宽度的相对量。

注 9：硬件像素。软件像素在 461 ppi 上实际是 1242 × 2208（http://bit.ly/curious-case-iphone6）。

视图控制器的可用尺寸类别基于以下三个因素：

- 屏幕尺寸
- 设备方向
- 视图控制器可用的部分屏幕（注意，当使用拆分的视图控制器展示主从控制器时，没有任何控制器可以访问整个屏幕）

在 interface builder 中设计视图控制器时，底部的布局工具条附近有尺寸类控制器，使用它选择一个类，如图 6-17 所示。

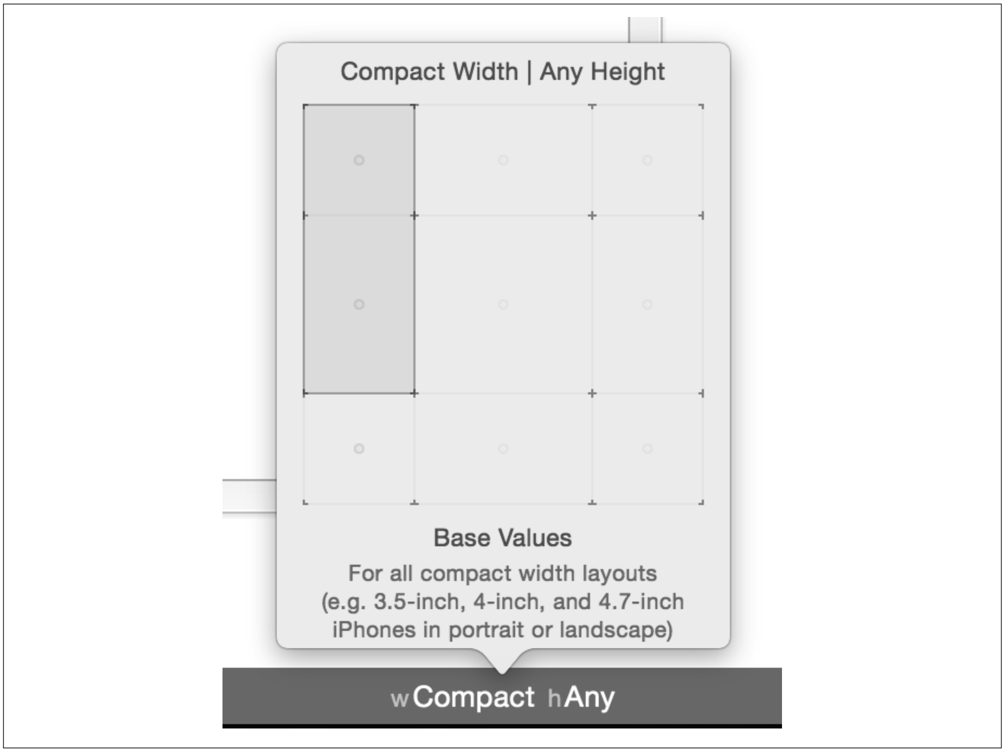


图 6-17：尺寸类选择

iOS 定义了两种尺寸类：紧凑型和普通型。当使用有限空间时，选择紧凑型的尺寸；使用广阔空间时，选择普通型。如图 6-17 所示，你可以选择一个水平尺寸类和垂直尺寸类来配置最终的 UI。

针对不同设备以及不同的方向组合，iOS 开发者库中的文章 iOS Human Interface Guidelines（https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/LayoutandAppearance.html#//apple_ref/doc/uid/TP40006556-CH54-SW1）提供了一些可选择的尺寸类的详细信息。图 6-18 将设备和方向与尺寸类进行了映射。

对于任何视图而言，你都能够更改所有可能影响最终位置和大小的参数。其中包括所有视图中的约束，以及所有可用的字体大小。图 6-19 展示了如何对某一特定的尺寸类添加约束。

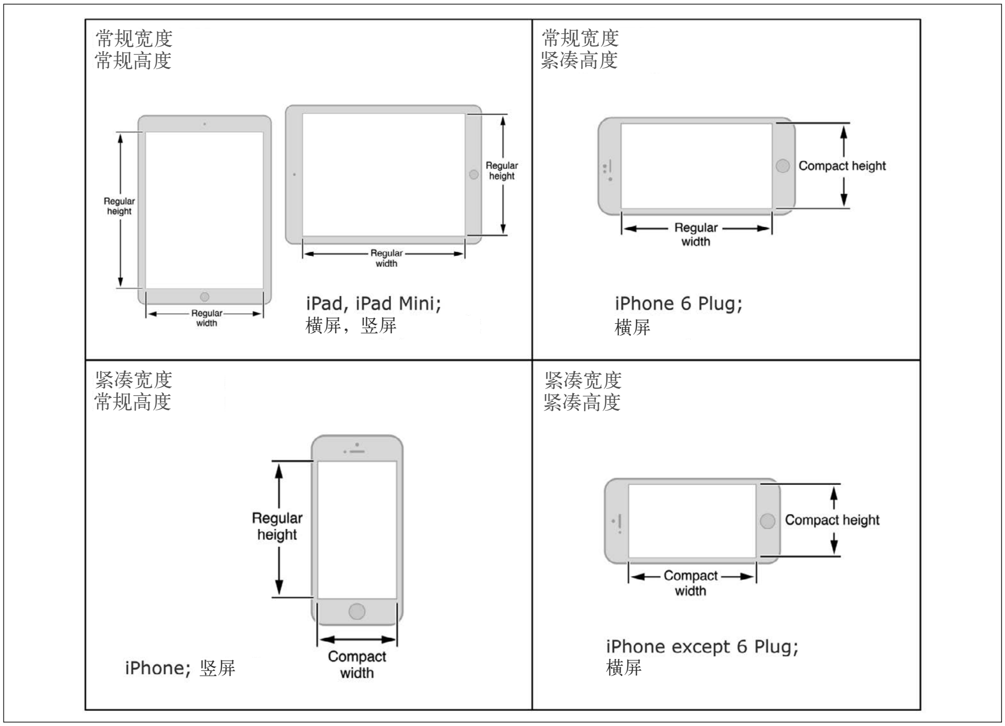


图 6-18：尺寸类与设备 / 方向的映射

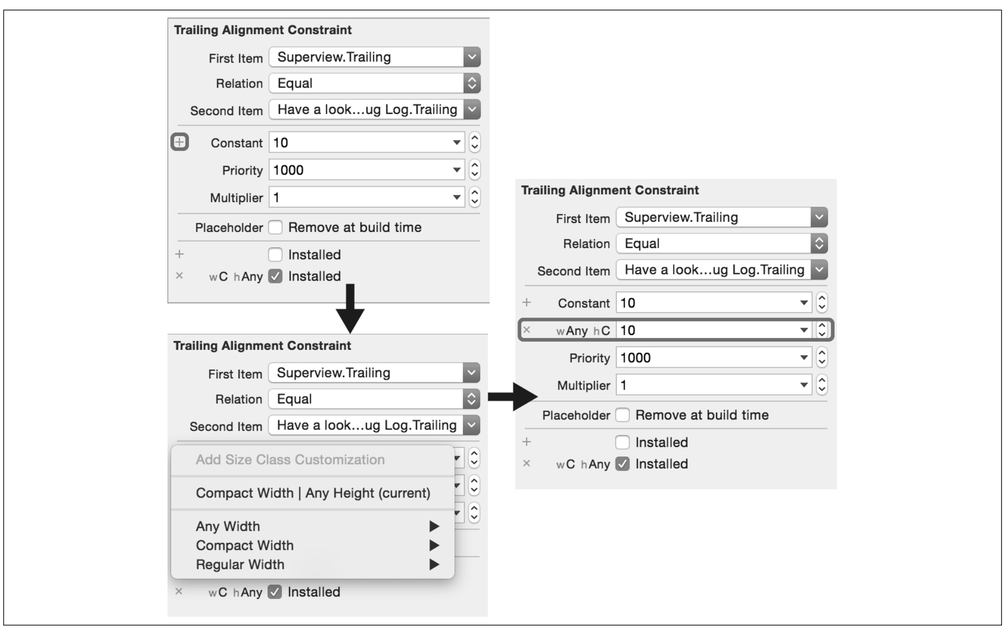


图 6-19：配置尺寸类——具体的参数

尺寸类提供了对 UI 进行分类的不同方法，无需再为简单场景（如方向）创建独立的布局。尽量在各个场景使用它们——建立基础设施的全部原因就是为了简化开发。

从性能角度看，加载故事板或 XIB 文件只有极少的影响。比较好的部分是，这些基于类的约束对最终的 XIB 文件的贡献只是一小部分。此外，如果应用支持多个方向，那 XIB 文件只被加载一次，当方向发生变化时，这为应用提供了很大的援助。而且请注意，在应用运行时加载 XIB 文件是一次性消耗。

> 尺寸类需要自动布局。如果你因为性能原因不选择使用自动布局，那就不能使用尺寸类。

6.5 iOS 8中新的交互特性

iOS 8 推出了两个神奇的功能，可以让用户与应用交互：

- 交互式通知
- 应用扩展

以下两节将探索这些功能。我们先查看每个特性的基本设置，然后看看如何使用它们提供最佳的用户体验。

6.5.1 交互式通知

你可以使用交互式通知，从而允许用户提供一个针对输入的快速响应。

截止至 iOS 7，用户点击通知（或在锁屏上滑动）只会调起应用。后续的操作只能在应用中进行。在 iOS 8 中，开发者为用户提供了一些操作，这些操作是在通知中预定义的行为。

以用户滑动的那个通知为基础，确定将要在应用中执行的行为。在 `application:didFinishLaunchingWithOptions:` 方法被调用时，通知的值可以从 `launchOptions` 中获取到。

这一种方式很好，但却无法处理以下的两种情况：其一，某个通知期望用户能够响应，且同时存在多个可用的响应选择；其二，获取响应的速度比点击后启动应用、将用户引导至某一具体的视图控制器（是一个非常麻烦且耗时的过程）更快。

在 iOS 8 中，你可以为一个通知添加*类别*，这样就可以为该类别定制一个或多个*动作*。用户可以选择其中任何可用的动作。

下面的例子是使用交互式通知时的可能动作。

- 邮件：回复，标记为垃圾。
- 信息：提醒，回复。
- 在社交应用中评论信息：回复评论，点赞评论。
- 任务和提醒：稍后，标记完成。

图 6-20 显示了一个示例，在使用交互式通知时，开发者提供了用于作出回应的快速操作界面。此处的例子是一个提醒通知。当向左滑动时，向用户展示了两个操作选项：稍后和完成。如果用户选择稍后，则提醒通知过一段时间后会再次弹出。如果用户选择了完成，则任务被标记为完成。

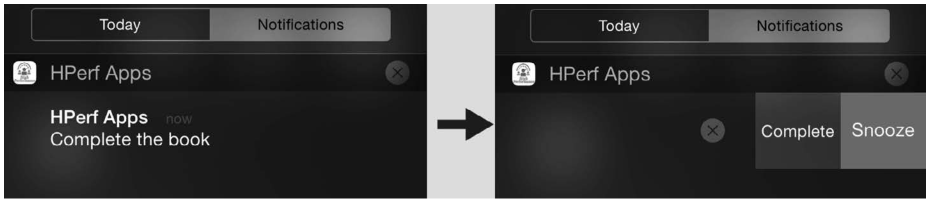


图 6-20：交互式通知——任务提醒

这样做的好处是，用户不需要打开应用就可以采取进一步的行动。在实现了 `UIAppDelegate` 协议的类中，`application:handleActionWithIdentifier:forLocalNotification:` 回调会处理通知。

在使用下拉横幅登录设备时，用户也可以响应通知。UI 有一些细小的差别，如图 6-21 所示。

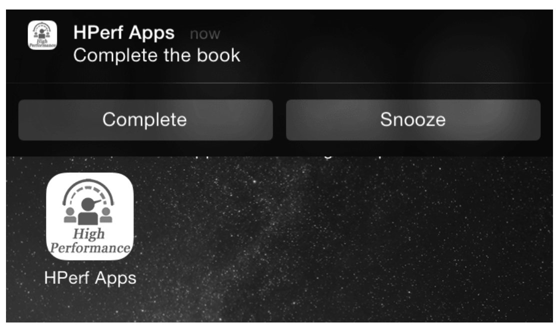


图 6-21：交互式提醒横幅

6.5.2　应用扩展

苹果开发者网站（https://developer.apple.com/app-extensions）对应用扩展进行了如下描述。

> 通过 iOS 8 和 OS X Yosemite，应用扩展使用户能够获取应用的功能和信息。例如，你的应用可以作为部件在“今日”屏幕上展示，在操作表中可以添加新的按钮，在 iOS 照片应用内提供照片过滤器，或显示一个新的全系统自定义键盘。使用扩展将应用的强劲能力放在用户最需要的地方。

iOS 8 引入了可以添加至应用的应用扩展。图 6-22 显示了一个 Xcode 菜单，在此可以将新的扩展添加至主应用中。

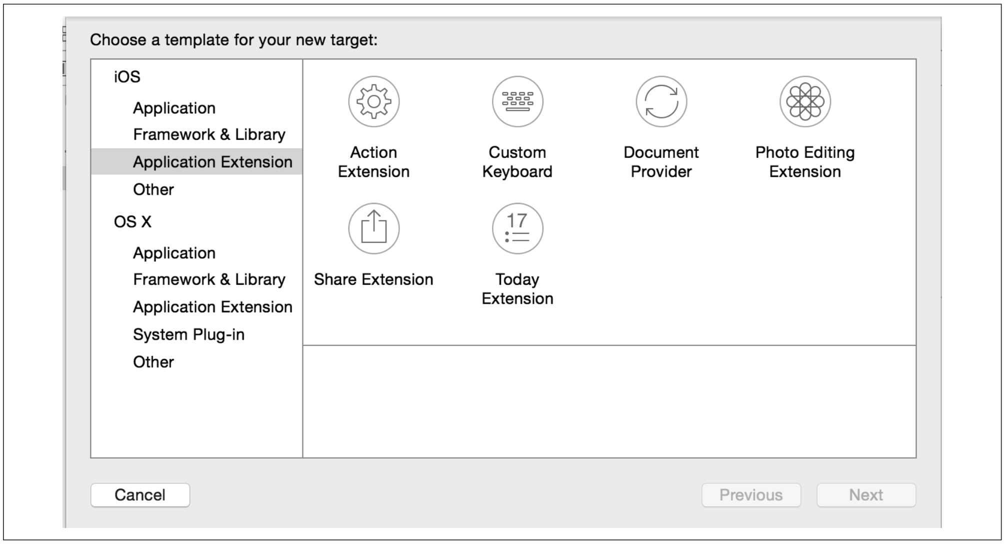


图 6-22：添加应用扩展的 Xcode 菜单

iOS 8 中可用的应用扩展如下。

- 今日（通常被称为窗口小部件）
  通知中心的“今日”视图可以帮助用户快速更新或快速完成某一任务（见图 6-23）。

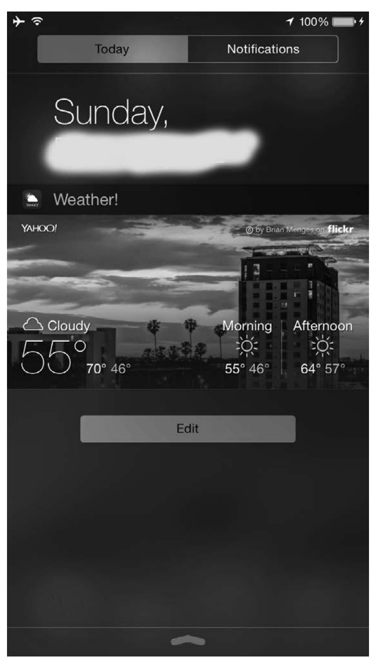


图 6-23：Yahoo 天气应用的窗口小部件

- 自定义键盘
  允许用户在任何应用中使用自己喜欢的键盘，包括搜索（见图 6-24）。自定义键盘最终能够自由地取代 iOS 系统键盘，并在所有应用中使用，这一行为无疑是伟大的。

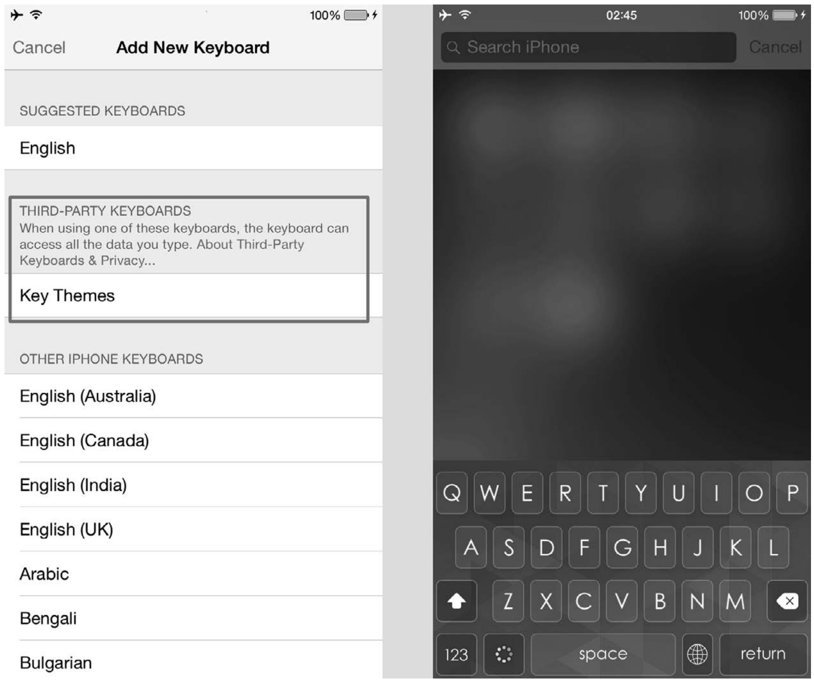


图 6-24：自定义键盘

- 分享
  允许用户更加无缝地跨应用共享数据。深层链接同样允许共享数据，但会强迫用户改为目标应用的上下文。
- 行动
  帮助用户查看或改变在主应用中发起的内容。例如，在收到视频时，你可能希望在喜爱的媒体播放器中观看，而不是在实际收到视频的电子邮件 / 消息应用中观看。
- 照片编辑
  允许用户编辑照片应用中的照片或视频。所以，你现在可以下载一个像 Adobe Photoshop Express 的应用来编辑所有的照片，而无需在该应用中手动打开要被编辑的照片；你现在可以从照片中启动编辑器了。
- 文档提供者
  允许其他应用访问自己应用所管理的文件。对某一类特殊类型的文档而言，文档提供者扮演了本地仓库的角色，让用户将所有文件收集至一个地方。

这些扩展被当作副包打包至主应用中，由同一生命周期和应用开发的规则管理。

扩展特有的最佳实践仍在不断发展。可能需要经历两个完整的应用版本的发行，才能将模式和最佳实践进行合并。目前，大部分准则和各个 iOS 应用的准则没什么差别。

6.6 小结

对视图控制器生命周期有了深入的理解后，现在你可以调整应用的感知性能了。用户可能知道应用缓存数据是为了获得更快的加载时间，或是对网络消耗的重视，但如果最终的用户交互执行地很差，那这一切都变成次要的了。

在声明性 UI 和程序性 UI 之间做选择时，你应该以性能和可扩展性为依据。本章对常见视图的深入分析应该使你能明智地使用它们，而对于应用中的给定场景，自定义视图的替代方案也为你提供了相关的选择。

需要注意的是，当以 60 帧每秒的速率渲染或更新 UI 元素、展示动画时（例如，滚动或其他方式），你大约有 16 毫秒来执行所有需要的操作。这可能包括网络或磁盘 I/O、视图内容的更新、布局和最终的渲染。可以按照下述原则将任务拆分为子任务，这个原则是：在一个事件循环中，保证子任务在主线程上累积消耗的时间最短。

第 7 章

网络

在应用中使用网络是必不可少的，但减少网络延迟的方法却是有限的（例如，使用 CDN 或边缘服务器、使用 Protobuf 或数据压缩这样小型的 payload 字段格式），因此，你应该着手对网络条件进行最大程度的优化，并预先对不同的场景进行规划。

本章将重点关注影响整体延迟的因素，并讨论如何充分利用现有的信息来最大程度地提高性能。

7.1 指标和测量

在网络中完成的大多数工作是你无法控制的，因此确定衡量的标准非常重要。我们将在本章中讨论一些较为重要的性能相关指标。请注意，此处并非要列举出所有的指标，只是挑出在性能优化相关的测量中更为重要的一些指标。

图 7-1 展示了典型的网络请求全景图。

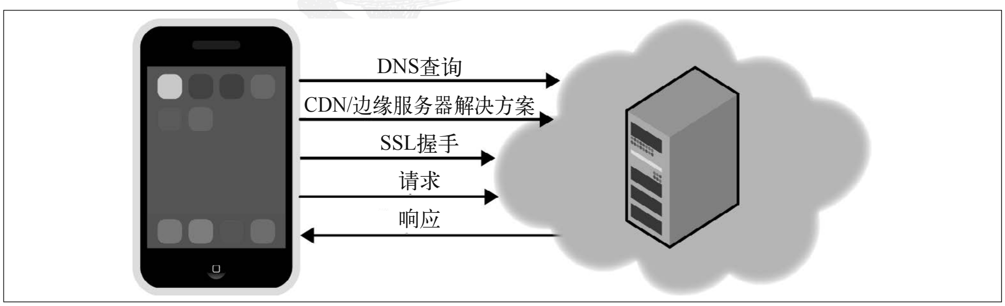


图 7-1：网络——设备至服务器

后续讨论的大致结构是：先进行相关指标的说明，然后是一个或多个示例，最后是最佳实践。

7.1.1　DNS查找时间

发起连接的第一步是 DNS 查找。如果你的应用严重依赖网络操作，DNS 的查找时间会使应用变慢。在一个关于两个位置的统计样本中，从加利福尼亚州的森尼韦尔市访问 www.google.com 时，DNS 查找时间是 2846 毫秒，而从印度的新德里访问时只花费了 34 毫秒（见图 7-2）。

查找时间与主 DNS 服务器的性能成函数关系。最终的连接时间与追踪到目的 IP 地址的路由成函数关系。

使用内容分发网络（content delivery network，CDN）将延迟最小化是一种常见的做法。在图 7-2 中，你应该注意到 www.google.com 在两个地点解析的 IP 地址是不同的。在森尼韦尔市，它被解析成了美国的一台服务器，而在新德里，它被解析成了印度的服务器。但因为 DNS 会为每一个独有的子域名进行查找，所有，拥有多个 CDN 主机名会导致应用的速度变慢。

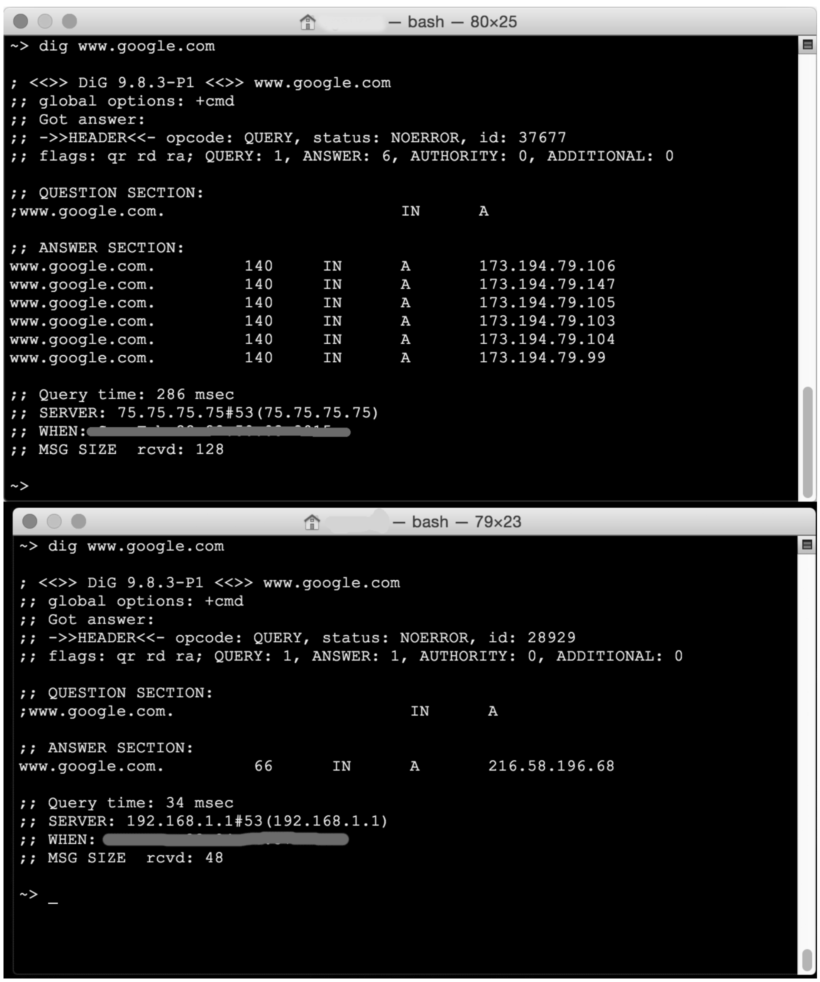


图 7-2：www.google.com 的 DNS 查询时间——加州的森尼韦尔市（上）和印度的新德里（下）

为了最大限度地减少 DNS 查询时间所产生的延迟，你应该遵循以下的最佳实践。

- 最小化应用使用的专有域名的数量。按照路由的一般工作方式，多个域名是不可避免的。最好是能做到以下几点：

  (1) 身份管理（登录、注销、配置文件）
  (2) 数据服务（API 端点）
  (3) CDN（图片和其他静态人工产品）

  有可能需要其他域名（例如，用于提供视频、上传检测数据、具体的子数据服务、广告投放，甚至是国家特定的全球本地化）。如果子域名数量上升至两位数，那么势必会引发担忧。

- 在应用启动时不需要连接所有的域名，可能只需要身份管理和初始画面所需的数据。对于后续的子域名，尝试更早地进行 DNS 解析，也被称为 DNS *预先下载*。为实现此操作，你可以参考以下两点。

  如果子域名和主机在控制范围内，你可以配置一个预设的 URL，不返回任何数据，只返回 HTTP 204 的状态码，然后提前对该 URL 发起连接。

  第二个方法是使用 `gethostbyname` 执行一个明确的 DNS 查找。然而，针对不同的协议，主机可能会解析至不同的 IP，例如，HTTP 请求可能会解析至一个地址，而 HTTPS 会解析至另一个地址。虽然不是很常见，但第 7 层的路由可以根据实际的请求解析 IP 地址，例如，图像是一个地址，视频是另外一个地址。鉴于这些因素，在连接之前解析 DNS 经常是无用的，对主机进行伪连接会更有效。

7.1.2 SSL握手时间

为了安全起见，可以假设应用中所有的连接均是通过 TLS/SSL 的（使用 HTTPS）。HTTPS 在连接开始时，先进行 SSL 握手，SSL 握手主要是验证服务器证书，同时共享用于通信的随机密钥。这一操作听起来简单，但是却有很多步骤，还会耗费较多时间（见图 7-3）。

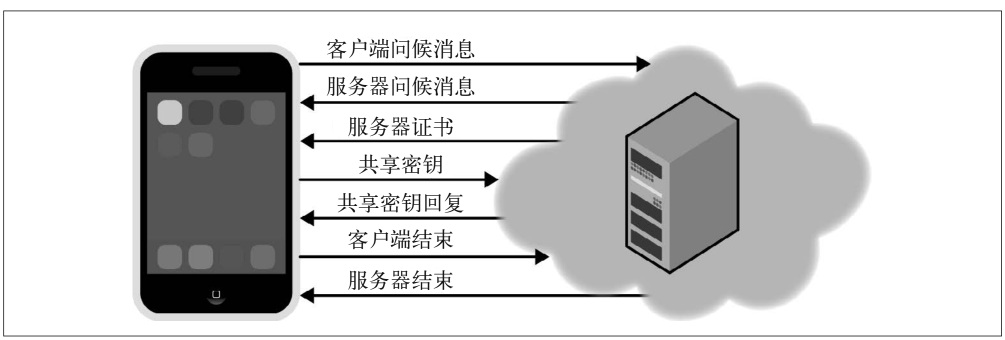


图 7-3：SSL 握手

你应该遵循以下的最佳实践。

- 最大程度地减少应用发起的连接数。因此，也需要减少应用连接的独有域名的数量。

- 请求结束后不要关闭 HTTP/S 连接。
  为所有的 HTTPS 请求添加头 `Connection: keep-alive`。这确保了同样的连接在下一次请求时可以复用。
- 使用域分片。如此一来，虽然连接的是不同的主机名，你也可以使用同一个 socket，只要它们解析为相同的 IP，可以使用相同的证书（例如，在通配符域）就行了。
  域分片在 SPDY 及其后续版本——HTTP/2（https://http2.github.io）——中是可用的。你需要一个支持上述任意一种格式的网络库。

> iOS 9 对 HTTP/2 有原生的支持。
> 对于 iOS 8 和之前的版本，你需要一个第三方库，如 Twitter 的 CocoaSPDY。你可以从 GitHub（https://github.com/twitter/CocoaSPDY）上获取。通过引入 CocoaPod `CocoaSPDY` 进行使用。

7.1.3 网络类型

由于用户逐步丢弃了桌面设备，这也就放弃了总是处于连接状态的高速带宽网络，转而使用有同等质量的 WiFi 网络或使用间歇连接的带宽可变的移动网络。更有挑战的场景是，用户是移动的。当设备在移动信号塔之间切换时，网络和质量也会发生变化。一个设备可能在任何时刻从 LTE 网络切换到 GPRS 或进入无信号区域。对于这种情况，人们往往束手无策。

> **主机的可到达性**
>
> 你可以使用苹果公司的可到达性库（https://developer.apple.com/library/ios/samplecode/Reachability/Introduction/Intro.html）或使用 Tony Million 对该库的简易替换（https://github.com/tonymillion/Reachability），主机的可到达性发生变化时，替换的库支持回调的调用。

> 如果设备闲置超过几秒（具体的值是不确定的，这里的几秒也可能变成几分钟），网络无线电可能已经关闭，这将导致无线资源控制器发生额外的几百或几千毫秒的延迟。
> 先确定主机的可到达性，从而确保应用具备处理此种场景的能力。

一般情况下，iPhone 和 iPad 可以使用下列任何网络连接到互联网。

- WiFi
  如果 WiFi 网络是私有网络（如家庭或办公室连接），那么你可以期望连接至互联网的网络是持续且质量较好的。
  但是，处于 WiFi 网络中并不能保证一定连接上了互联网。例如，当设备连接到某一公共热点（例如，在旅馆或购物商场）时，如果用户没有成功地提供适当的凭证，那么将无法访问互联网。

即使设备已经连接至互联网，也可能有一些限制，比如可连接的域名或端口限制。举个例子，www.google.com 或 www.yahoo.com 域可能是允许的，但 mail.google.com 或 mail.yahoo.com 却可能无法使用。

- 4G：LTE、HSPA+（高速的数据网络）
  这些是最新一代的数据网络。在第一个真正的业务数据发送前，一般会有 100~600 毫秒的延迟。这些网络以亚毫秒的间隔动态地创建无线相关的资源，并且爆发性地发送数据。理论上来说，速度会从 100Mbps 浮动至 1Gbps。对于高速率移动的通信，如在汽车或火车上，速度可能会在 100Mbps；对于低速率移动的通信，如行人或静止的用户，速度可能会达到 1Gbps。
- 3G：HSDPA、HSUPA、UMTS、DMA2000（中等速度的数据网络）
  这些是上一代的数据网络，但使用频率可能比 LTE 更频繁。
  3G 的速度可能会从 200Kbps 变化至超过 50Mbps。双向的传输速度可能并不对称。HSDPA 具有较高的下行速度，HSUPA 具有较高的上行速度。
- 2G：EDGE、GPRS（低速的数据网络）
  20 世纪 90 年代的网络仍然没有消亡。这些都是初始数字网络（第 1 代网络使用模拟信号），提供了较低的带宽。EDGE 理论上具有 500Kbps 的极限速度，而 GPRS 最高只能达到 50Kbps。

可到达性库可以给出访问主机的详细网络信息。使用此信息来确定传输内容的类型（例如，文本、图像或是视频），以及传输的各种项目的大小，等等。

0.2%的数据传输；46%的功耗！

2011 年，密歇根大学和 AT & T 发布了“移动应用性能资源使用情况”（http://mobilityfirst.winlab.rutgers.edu/documents/mobisys11.pdf），这篇研究性论文分析了移动应用的网络使用和功耗效率。

论文探讨了潘多拉公司，将其在移动网络的间歇性网络传输的低效率问题作为经典案例研究。虽然问题已经修复了，但案例分析还是值得阅读的。

当播放一首歌曲时，应用会将这首歌全部下载，这是正确的行为：下载尽可能多的数据，让无线电关闭的时间尽可能长。

但是，在传送之后，应用会每 60 秒定期地发送检测事件。这些事件仅占传输总字节的 0.2%，但却占了应用总功耗的 46%。

事件的数据通常是比较小的，但因为无线电在较长的时间都会保持激活状态，所以它将应用的电量消耗增加了一倍。

通过将这些数据分发至一些请求之中，或在无线电已经处于激活状态时再发送数据，就可以消除不必要的能量拖尾，实现更高的电源效率。

为确保你的应用不会成为类似案件研究的一部分，在开发以网络为中心的应用时，你可以

遵循以下的最佳实践。

- *设计时考虑不同的网络可用性。*在移动网络中，唯一不变的是，网络可用性是多变的。对于流媒体，最好选择 HTTP 实时流或任何可用的自适应比特率流媒体技术，这些技术可以在某一时刻针对可用带宽进行动态切换，切换至当前带宽的最佳流质量，从而提供流畅的视频播放。
  对于非流媒体内容，你需要实现一些策略，确定在单次拉取时应该下载多少数据，并且数据量必须是自适应的。例如，你可能不希望在最新一次更新时，一次拉取所有的 200 封新邮件。你可以先下载前 50 封邮件，再逐步下载更多邮件。

  同样，在低速网络时，不要打开视频自动播放功能，这可能会花费用户很多钱。

  对于自定义的非流媒体数据拉取，要保持对服务器的关注。让客户端发送网络特征数，服务器决定返回的记录条数。这样一来，你可以在不发布应用新版本的情况下进行适应性改变。

- *出现失败时，在随机的、以指数增长的延迟后进行重试。*
  例如，第一次失败后，应用可能会在 1 秒后重试。第二次失败时，应用在 2 秒后重试，接着是 4 秒的延迟。不要忘记对每个会话设置最多的自动重试次数。

- *设立强制刷新之间的最短时间。*当用户明确要求刷新时，不要立即发出请求。相反，检查是否已经存在一个请求，或当前请求与上次请求的时间间隔是否小于阈值。如果满足上述条件，则不要发送此次请求。
- *使用可到达性库发现网络状态的变化。*如图 7-4 所示，使用指示条向用户展示不可用的状态，毕竟设备没有网络连接并非你的错。通过让用户了解潜在的连接问题，可以避免你的应用受到指责。

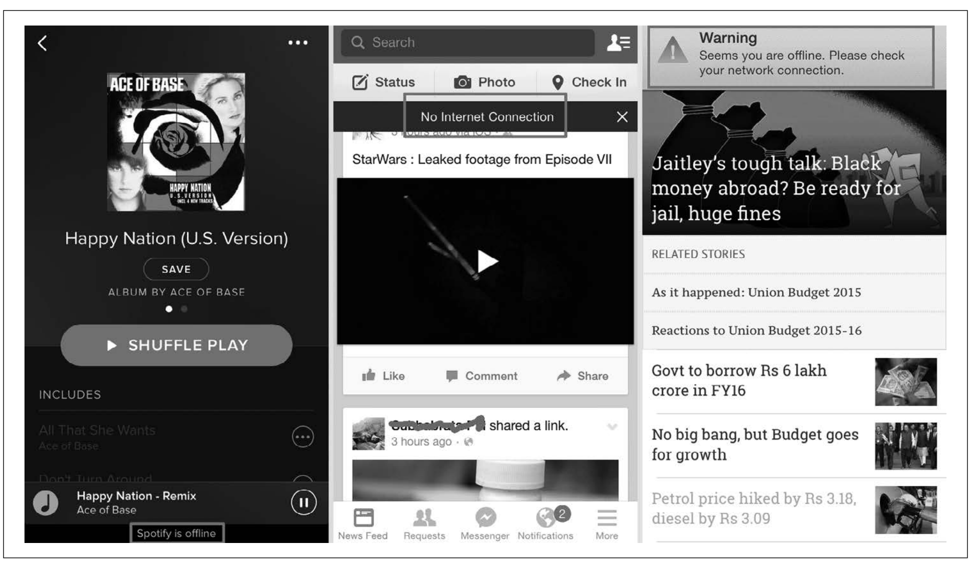


图 7-4：Spotify、Facebook 和 TOI 针对离线网络状态的指示条

- *不要缓存网络状态*。不论是通过触发请求时的回调来获取状态，还是在发送请求之前显式地检查状态，要始终使用网络敏感度高的任务的最新值。
- *基于网络类型下载内容*。如果想要展示一个图像，不用总是下载原始的、高质量的图像。应该始终下载和设备适配的图像——iPhone 4S 所需的图像尺寸和第三代 iPad 所需的差别很大。
  如果应用有视频内容，最好有一个与之关联的预览图像。如果应用支持自动播放功能，在非 WiFi 网络中只展示预览图像，因为自动播放会花费用户很多钱。

  此外，针对图像、音频和视频等，提供一个关闭自动下载或关闭自动播放功能的选项。图 7-5 展示了 WhatsApp 应用中相关的设置。

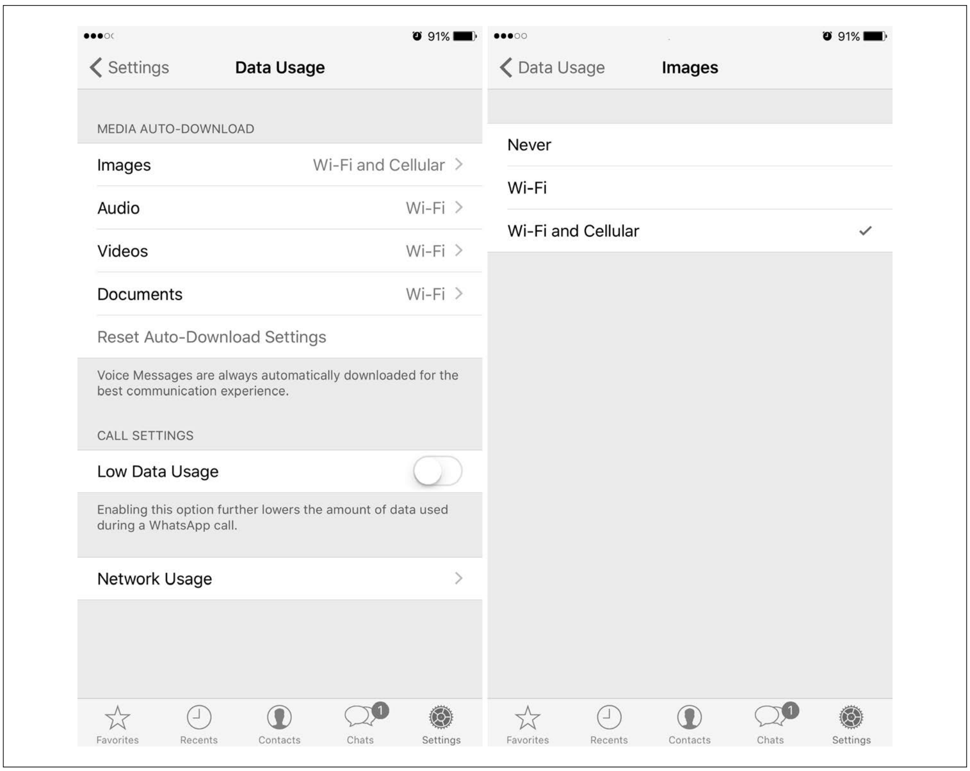
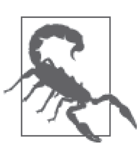


图 7-5：WhatsApp 中对图像、音频和视频内容下载的可选设置

- *乐观地预先下载*。在 WiFi 网络中预先下载用户在后续时刻需要的内容。随后就可以使用缓存内容了。最好分次下载内容，在使用之后关掉网络连接，这有助于节省电量。



预先下载永远都是争论的焦点。在下载最少数据和获取最近可能需要使用的所有内容之间，人们总会存在激烈争执。

没有黄金法则可循。这在很大程度上取决于平均数据的大小、完成的下载数、预期的使用模式和网络条件。如果网络不断变化，而且你需要执行最小的数据传输，看看是否可以分批处理这些请求。

- 如果适用，当网络可用时，支持同步的离线存储。通常情况下，网络缓存就足够了。但如果需要更多的结构化数据，使用本地文件或 Core Data 会是一个较好的选择。
  对游戏来说，缓存最近一级的详细信息。对邮件应用来说，存储一些带有附件的最新电子邮件是一个不错的选择。

  根据不同的应用，你可能会允许用户创建新的离线内容，离线内容会在网络连接可用时和服务器进行同步。例如，在邮件应用中编写新邮件或回复某封邮件时；在社交应用中更新资料图片时；拍摄将要上传的照片或视频时。

  总是将网络和通信与 UI 解耦。如果应用可以进行离线操作，那么就通知用户离线操作是可行的，否则就通知用户不能进行离线操作。不要让用户已经开始和应用进行交互之后，获取不到返回值。这会是一个糟糕的用户体验。

> 不要为金融、银行、股票交易或其他需要和服务器同步的应用增加离线交易的选项，因为更新的数据可能在离线模式下不可用。

图 7-6 展示了离线模式下的 Facebook 和 E*Trade 应用。Facebook 应用通知用户网络不可用，但允许发布评论或状态更新。当网络可用时，上述内容会在后续进行同步。在 E*Trade 应用中，用户也可以与应用进行交互，但当查找某一股票的报价时，应用会进入死胡同，这会导致糟糕的用户体验。

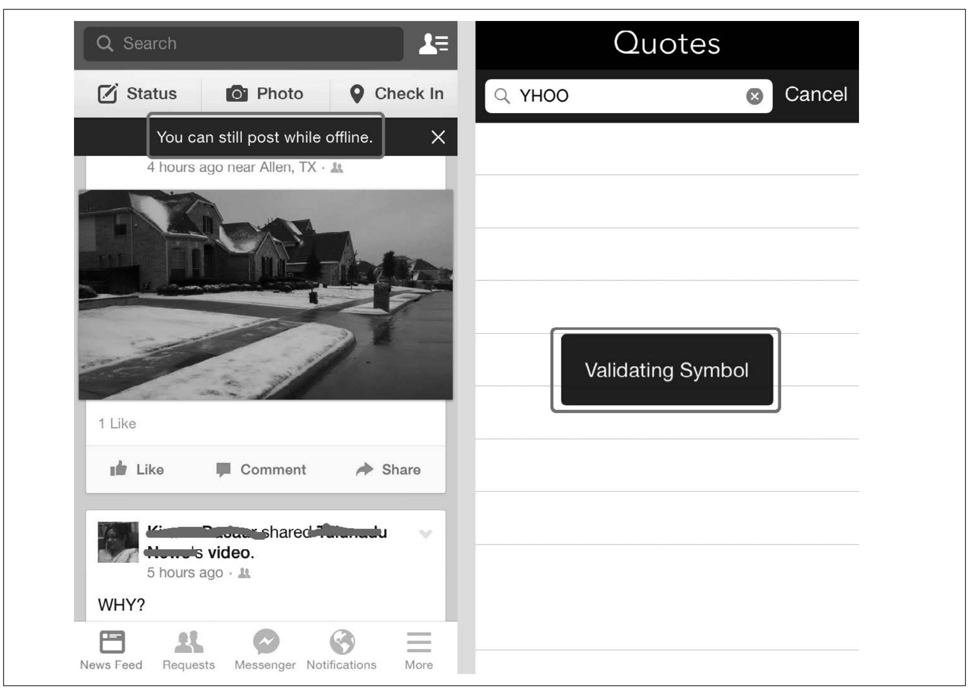


图 7-6：离线模式下的应用——Facebook（左）和 E*Trade（右）

需要注意的是，网络条件总是会超出应用的控制，但在这些限制范围内提供的用户体验却是受应用控制的。要将可用选项做到最好，这些选项包括离线存储、网络可到达性、网络类型、执行（或不执行）网络操作，通知（或不通知）用户相关的信息。

7.1.4 延迟

延迟是指从服务器请求资源时，在网络传输上花费的额外时间。设置用于测量网络延迟的系统是很重要的。

网络延迟可以通过使用请求过程中花费的总时间减去服务器上花费的时间（计算和服务响应）来测量：

```
Round-Trip Time = (Timestamp of Response - Timestamp of Request)
Network Latency = Round-Trip Time - Time Spent on Server
```

花费在服务器上的时间可以由服务器来计算。对客户端而言，往返的时间是准确可用的。服务器可以将花费的时间放在响应的自定义头部，然后客户端就可以用来计算延迟了。

例 7-1 给出了计算延迟的示例代码。该代码假设响应包含了自定义头部 `X-Server-Time`，这个时间以毫秒为单位，包含在服务器上花费的时间。

例 7-1 计算网络延迟

```
//server - NodeJS
app.post("/some/path", function(req, res) {
    var startTime = new Date().getTime();
    //处理
    var body = processRequest(req);
    var endTime = new Date().getTime();
    var serverTime = endTime - startTime;
    res.header("X-Server-Time", String(serverTime));
    res.send(body);
});

//client - iOS app
-(void)fireRequestWithLatency:(NSURLRequest *)request {

    NSDate *startTime = nil;
    AFHTTPRequestOperation *op =
        [[AFHTTPRequestOperation alloc] initWithRequest:request];
    [op setCompletionBlockWithSuccess:^(AFHTTPRequestOperation *op, id res) {

        NSDate *endTime = [NSDate date];
        NSTimeInterval roundTrip = [endTime timeIntervalSinceDate:startTime];
        long roundTripMillis = (long)(roundTrip * 1000);

        NSHTTPURLResponse *res = op.response;
        NSString *serverTime = [res.allHeaderFields objectForKey:@"X-Server-Time"];
        long serverTimeMillis = [serverTime longLongValue];

        long latencyMillis = roundTripMillis - serverTimeMillis;
```

```
        } failure:^(AFHTTPRequestOperation *op, NSError *error) {
            //处理错误。如果需要,向用户展示错误
        }];

        startTime = [NSDate date];
        [op start];
    }
```

例 7-1 中的代码关于网络延迟的表达基本准确，但它包括了服务端线路上刷新数据花费的时间，以及在客户端解析响应花费的时间。如果可以分离出来，它将提供真实的网络延迟时间，包括任何设备开销。

如果你有数据来分析任何模式下的延迟，还需跟踪下列数据。

- 连接超时
  跟踪连接超时的次数是非常重要的。根据网络质量（较薄弱的基础设施或较低的容量），该指标会提供详细的地理区域分类，网络质量将反过来帮助规划同步时间的传输。例如，同步会在短时间间隔传输，比如几分钟，而不用在某一个特定时间跨时区同步。

- 响应超时
  捕捉连接成功但响应超时的数量。这有助于根据地理位置和日期、年份的时间来规划数据中心的容量。

- 载荷大小
  请求以及响应的大小完全可以在服务器端进行测量。使用此数据可以识别任何可能降低网络操作速度的峰值，并确定一些可用选项：通过选择合适的序列化格式（JSON、CSV、Protobuf 等）减少数据占位，或者分割数据并使用增量同步（例如，通过使用小的批量大小或在多个块中发送部分数据）。

较差网络容量的最大化

我曾经开发过一个神奇的体育应用，当时的工程师团队渐渐注意到应用的延迟变得更长，超时（连接和响应超时）变得更多了。我们还发现，服务器通常会发送超过 200KB 的压缩 JSON 数据作为初始开销，因此我们必须在比赛开始前约 20 分钟内做这件事情。

在比赛当天晚上，现场有超过 10 000 个用户连接至一个基站，总共有 50 000~80 000 个用户，造成了移动数据网络的阻塞。

虽然不能做任何事情改善连接，但我们使用了一些技巧来改善体验。开始时，我们向设备发送了推送通知。在开始前几个小时发送出去的第一个推送通知用于询问用户是否将要去比赛场。并非所有用户都回应了，但相当多的用户回应了（我们采用了游戏化来激励）。这不仅提供了估算流量的数据，更重要的是明确了哪些用户需要通知。

第二个推送通知只发送给了表示将要前往比赛场的用户。这个推送通知是在比赛的前 20 分钟分批发送出去的。如果球场有 1000 个用户，其中的 100 个用户会在初始的两分钟收到通知，下 100 个用户会在接下来的 2 分钟收到通知，并依次类推。

通知将唤醒应用，通过使用地理位置来决定是否获取数据。现在不再是 1000 个人同时连接，连接将在以 100 人为一组的用户组中同时进行。

显而易见，你不能指望每个用户立即打开应用，但一个推送通知将有助于唤醒应用，并让它同步数据。与应用进一步交互时会变得更加顺畅。

7.1.5 网络API

在执行任何网络操作时，重要的是你选择的 API。

iOS 的早期版本提供了 `NSURLConnection` 来执行网络请求。应用的开发人员需要管理连接池，并处理应用后台、中断以及请求的恢复。

`NSURLSession` 于 iOS 7 推出，现在应该是执行任何网络操作的实际选择。使用 `NSURLSession` 具有以下优点。[1]

- `NSURLSession` 对于放入其中的相关请求而言是一个可配置的容器。例如，对服务器的所有请求都可配置成始终包含访问令牌。
- 你可以得到后台联网的所有好处。这有助于提高电池寿命、支持 UIKit 的多任务、使用与过程传输相同的委托模型。
- 任何网络任务都可以暂停、停止并重新启动。与 `NSURLConnection` 不同，此处并不需要是 `NSOperation` 的子类。
- 你可以继承 `NSURLSession` 来配置会话，以便在每个会话的基础上使用专用存储（缓存、cookie jar 等）。
- 当使用 `NSURLConnection` 时，如果遇到了身份验证问题，问题会在任何一个请求中返回，你无法明确知道哪个请求遇到了这个问题。使用 `NSURLSession`，委托会处理身份验证。
- `NSURLConnection` 有一些基于块的异步方法，但委托不能使用它们。当发起一个请求时，要么成功，要么失败，即使它需要身份验证。
- 使用 `NSURLSession`，你可以采取一种混合的方法，这也就是说，你可以使用基于块的异步方法，还可以设置委托以处理身份验证。

7.2 应用部署

随着对这些指标的统计，你可以更好地规划应用的部署。这不仅包括服务器、服务器的位置和容量，还包括客户端，以及如何在给定的场景下获得最好的。

在本节中，我们将从网络的角度研究端到端应用的重要组件——在掌控之中的那些组件。

注 1：使用的有关提示请参见 Ken Toh 的“NSURLSession Tutorial: Getting Started”（http://www.raywenderlich.com/51127/nsurlsession-tutorial）。

7.2.1　服务器

在查看网络延迟的地域分布时，我们可以使用这个信息为数据中心选择适当的位置。如果使用托管的数据中心提供商，不妨选择有多个地理位置的，如 Amazon AWS 或 Rackspace Cloud。如果你有自己的数据中心，那么应该确保它们在地理上是分散的。

无需多想，服务器应该安装在多个位置，这样你可以更好地服务本地内容。

以下是一些应该遵循的最佳实践。

- 使用多个数据中心，让服务器在地理上分散开来，更贴近用户。
- 使用 CDN 提供静态内容，如图像、JavaScript、CSS、字体等。
- 使用接近的边缘服务器（http://serverfault.com/a/67489）来提供动态内容。
- 避免使用多个域名（DNS 查询时间可能会很长，这会降低用户体验）。

注意，第二点和第四点是互斥的，你需要进行权衡。当使用 DNS 时，最大限度减少 DNS 查找时间的信息请参见 7.1.1 节中讨论的最佳实践。

7.2.2　请求

为了恰当地设置网络，正确地配置 HTTP/S 请求很重要。你应该遵循以下的最佳实践。

- 不要为每一个操作单元都进行一次请求，使用批量请求。即使必须实现多个后端子系统来完成，但是合并批量请求会带来较大的性能提升，所以还是值得的。
  客户端可以向多个后端发送多路复用的请求，而服务器可以使用多部分 / 混合回复作为回应。客户端将对回复进行复用。图 7-7 展示了实现这一功能的概要图。

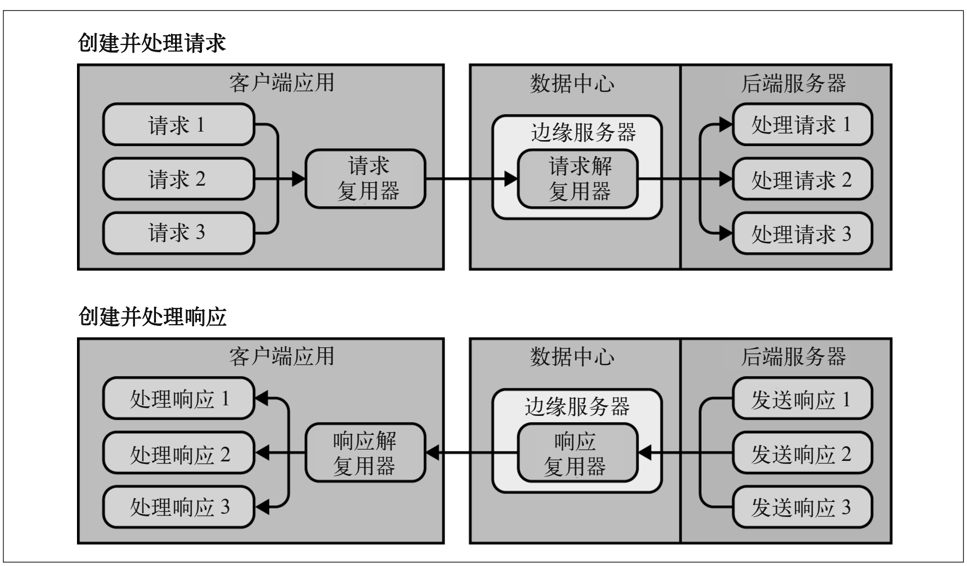


图 7-7：请求的多路复用系统

- *使用持续的 HTTP 连接*，该连接也被称为 HTTP 长连接。它们有助于最大限度地减少 TCP 和 SSL 握手的消耗，同时也减少了网络拥塞。
  或者使用 WebSockets。Square 的 SocketRocket（https://github.com/square/SocketRocket）这样的库可以协助人们在 iOS 上使用 WebSocket。
- 在任何可以的情况下都使用 HTTP/2（https://http2.github.io/）。通过单一的连接，HTTP/2 支持 HTTP 请求的真正复用；如果请求解析为一个 IP 地址，那么 HTTP/2 会将跨越了多个子域的请求聚集到一起；HTTP/2 还支持报头压缩等。使用 HTTP/2 的好处是巨大的。最好的是，就消息结构而言，该协议仍旧保持不变，依然包括头部和主体。
- *使用 HTTP 缓存头设置正确的缓存级别*。对于想要下载的标准图像（如主题背景或表情），内容的有效期可以设置为较长的时间。这不仅保证了网络库在本地缓存它们，还保证了其他设备可以从在本地进行了缓存的中介服务器（ISP 服务器或代理）中受益。影响 HTTP 缓存的响应头是 `Last-Modified`、`Expires`、`ETag` 和 `Cache-Control`。

7.2.3 数据格式

选择正确的数据格式和选择网络参数一样重要。一些选择可能会使应用的性能产生很大的不同，比如对无损图像压缩使用 PNG 还是 WEBP。

如果你的应用是以数据为导向的，那么选择适合其传输的正确格式很关键。其他协议支持的功能也可以提供帮助。

需要注意的是，虽然使用了 SSL，但也可能会有安全问题。

在选择数据格式时，你应该遵循以下的最佳实践。

- *使用数据压缩*。当传送 JSON 或 XML 这样的文本内容时，这一点尤为重要。`NSURLRequest` 会自动给头部添加 `Accept-Encoding：gzip`、`deflate`，这样你就无需自己动手了。但这也意味着服务器应该承认头部，并使用适当的传输编码发送数据。
- *选择正确的数据格式*。不用多想，JSON 和 XML 这样冗长、人类可读的格式是资源密集型的——序列化、传输、反序列化会比使用自定义制作的、二进制的、机器友好的格式更耗费时间。此处不讨论媒体压缩（即图像压缩和视频编解码器），而是着眼于文本数据格式。
  原生应用最常选用的数据格式正好是 JSON 和 XML。唯一的原因是，web 服务 /API 是为 Web 编写的，并且用于移动端。

  但是，如果你还没有准备好，那就需要开始思考移动端了。前面提到的格式是很方便手工制作的，但对机器操作而言却是资源密集型的。最好从大小以及序列化 / 反序列化的角度综合来看，选择一个更优的格式。

  对于运输记录而言，最流行的二进制格式是 Protocol Buffers，也被人称之为 Protobuf。其他协议包括 Apache Thrift 和 Apache Avro。通常情况下，Protobuf 被认为要胜于其他，但很多东西还是取决于使用的数据类型。如果大部分数据都是字符串，那么你应该找方法优化它们的加载，因为它们不会通过任何格式被压缩。使用 `deflate`、`gzip` 或任何无损压缩算法进行压缩。

7.3 工具

奠定了正确的基础之后，你还需要从一些工具获得某些支持，从而让执行更加顺畅。

7.3.1 网络链接调节器

网络链接调节器在 iOS 设备以及模拟器中都是可用的。在设置应用中，你可以通过开发者菜单进行访问。图 7-8 显示了如何获取网络链接调节器和相关设置。

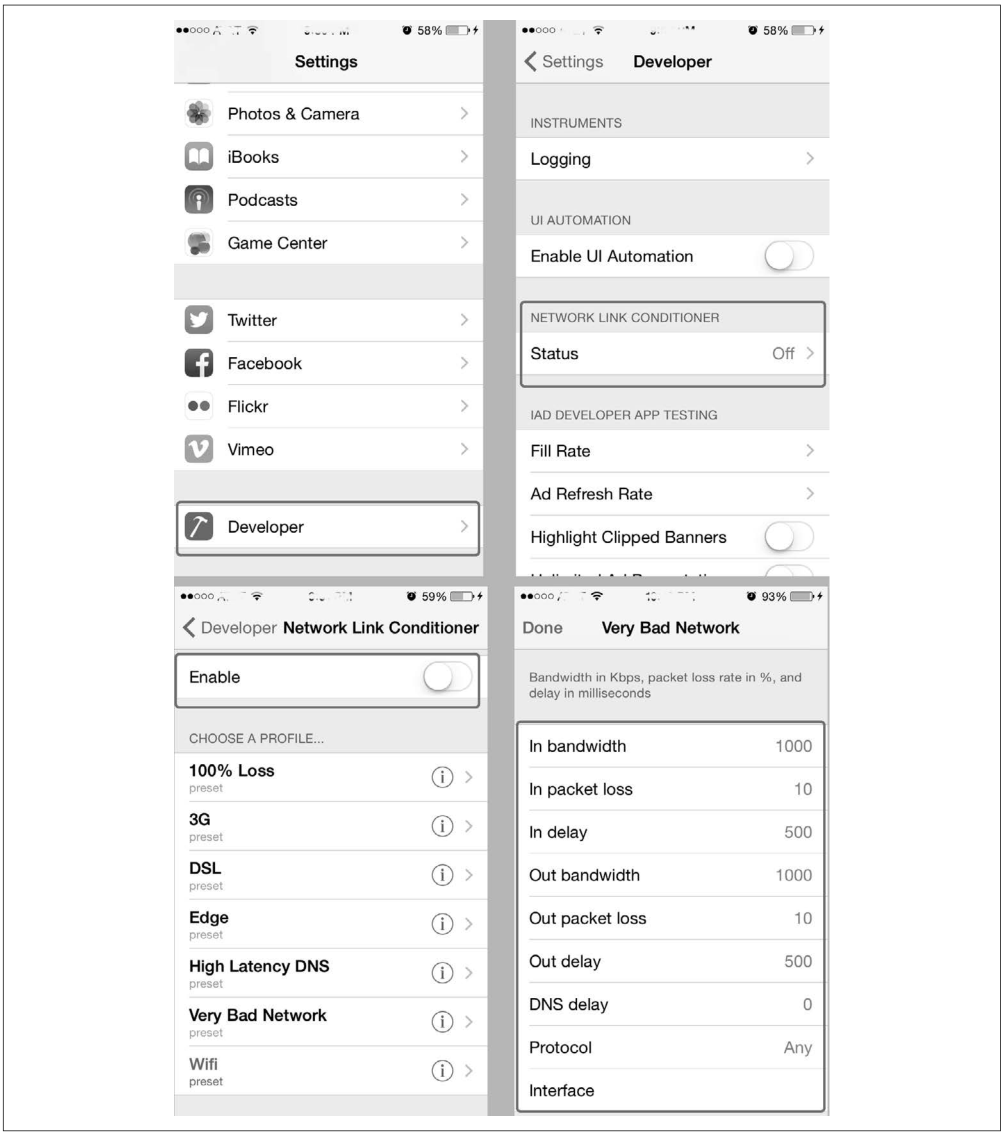


图 7-8：网络链接调节器的设置

你可以选择一个预定义的配置文件，或创建一个新的配置文件。通过控制重要的参数，网络链接调节器可以模拟不同的网络条件：

- 入站通信
  带宽、数据包丢失和延迟（响应延迟）
- 出站通信
  带宽、数据包丢失和延迟
- DNS
  查找延迟
- 协议 IPv4、IPv6，或两者
- 界面
  WiFi、移动，或两者

你应该使用网络链接调节器来测试应用在极端情况中的行为。你可能不会定期做这些测试，但在每一个新版本发布之前，至少应该测试一次。

7.3.2 AT&T应用资源优化器

虽然官方文档（http://developer.att.com/application-resource-optimizer/docs）指出，AT & T 的应用资源优化器（http://developer.att.com/campaigns/application-resource-optimizer）工具可以优化*移动 web 应用*的性能，但它也可用于本地应用。

要想使用此工具，需要将 iPhone/iPad 配置为开发和调试状态（使用 Xcode），这样你就可以看到前面讨论过的开发者菜单了，同时你还需要 Mac 上的管理权限。

你可以通过以下步骤启用开发者菜单。

(1) 将 iOS 设备连接到 Mac OS X 设备。
(2) 打开 Xcode。
(3) 导航到 Window → Devices.。
(4) 选择 iOS 设备并选择 " 用于开发 "。

应用资源优化器工具包括两个步骤：*数据收集*和*数据分析*。为了收集数据，请按照下列步骤操作。

(1) 导航到 Menu Bar → Data Collector → Start Collector。
(2) 在设备上运行应用。
(3) 导航到 Menu Bar → Data Collector → Start Collector。

通过对比一系列的最佳实践测试，数据分析器会评估从应用中收集到的数据。结果在总结屏幕中展示了出来（如图 7-9 所示）。

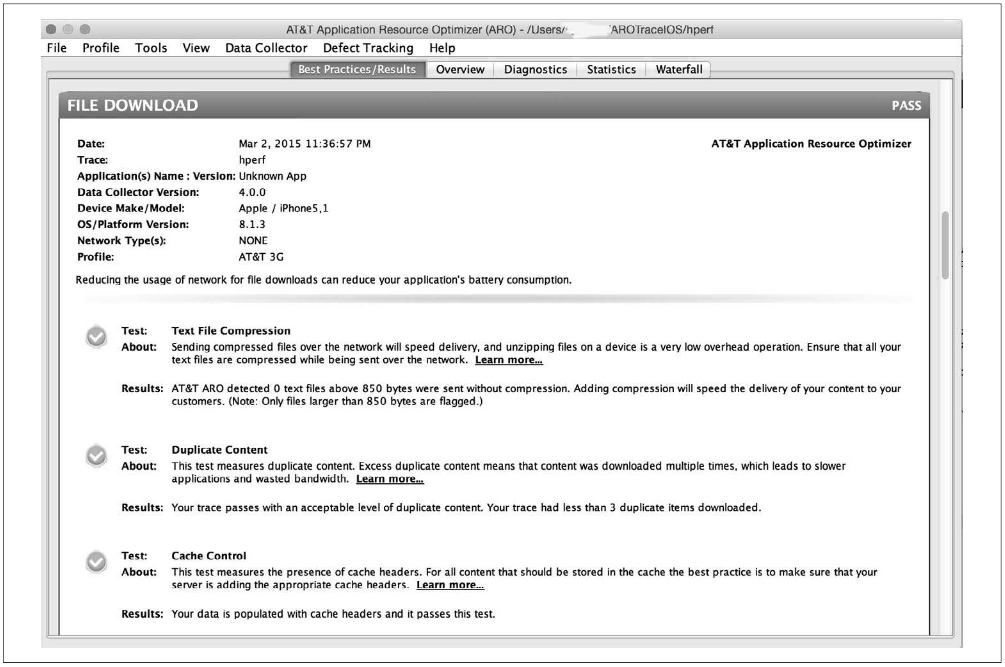


图 7-9：应用资源优化器数据分析器：总结

该工具在 ~/AROTraceIOS/{name}/ 中收集数据。在收集的总体数据中，以下是比较重要的：

- 设备的详细信息（型号、操作系统版本，屏幕尺寸等）
- 通过 pcap 接口的流量的详细信息
- 电池消耗

图 7-10 显示了应用资源优化器工具中的网络分析汇总图。

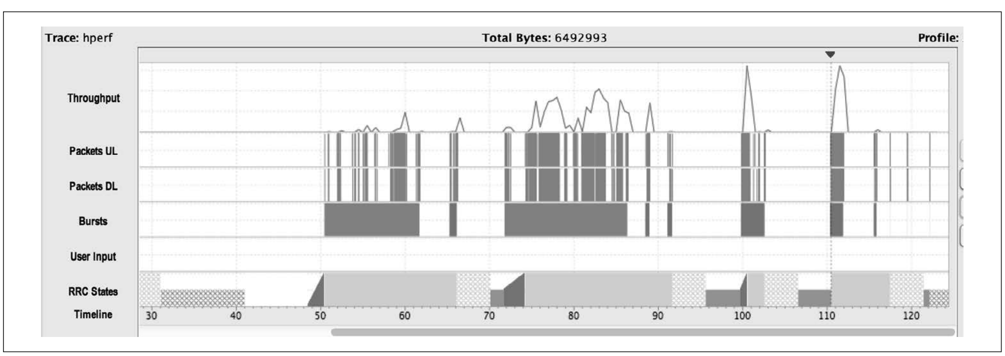


图 7-10：应用资源优化器数据分析器：网络用量的诊断

这种分析以一种易于理解的方式呈现出来，但可以通过编程的方式分析原始数据，从而获得更详细的报告，更重要的是，历史数据可用来分析以时间为轴线的性能变化。

7.3.3 Charles

Charles（http://www.charlesproxy.com）是一个非常强大的网络调试代理。你可以对其进行配置，以完成以下操作。

- 监视 HTTP 请求

  可以监视 HTTP 流量，包括请求、响应数据以及 HTTP 头。图 7-11 显示了来自 Facebook 应用的示例请求。图 7-12 显示了同一请求的响应。响应是 JSON 格式的，可以在不使用任何外部工具的情况下对其进行格式化。

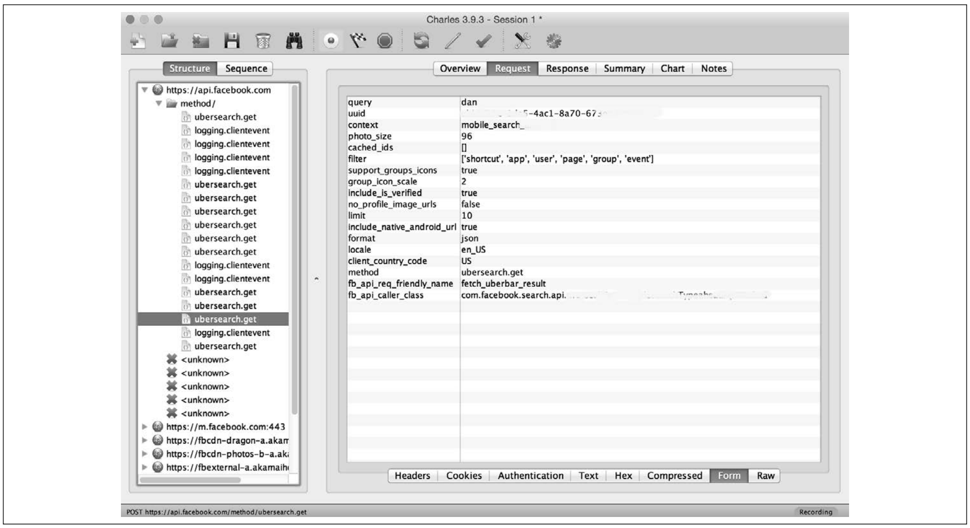


图 7-11：Charles：请求细节

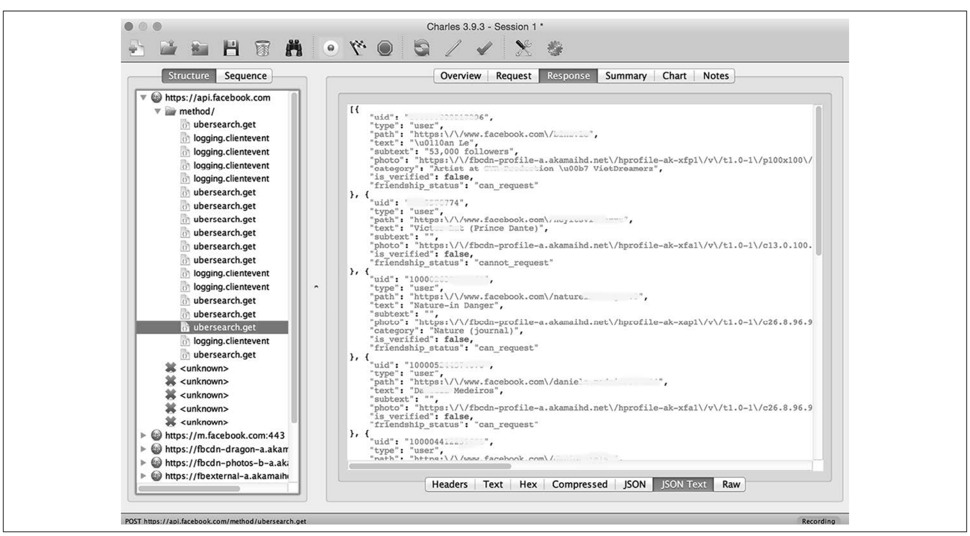


图 7-12：Charles：响应细节

- 监控 HTTPS 请求
  这需要设置证书。你可以创建自己的签名证书或使用网站提供的默认证书（http://charlesproxy.com/charles.crt），如图 7-13 所示。在设备上安装证书，就可以查看 HTTPS 请求了。

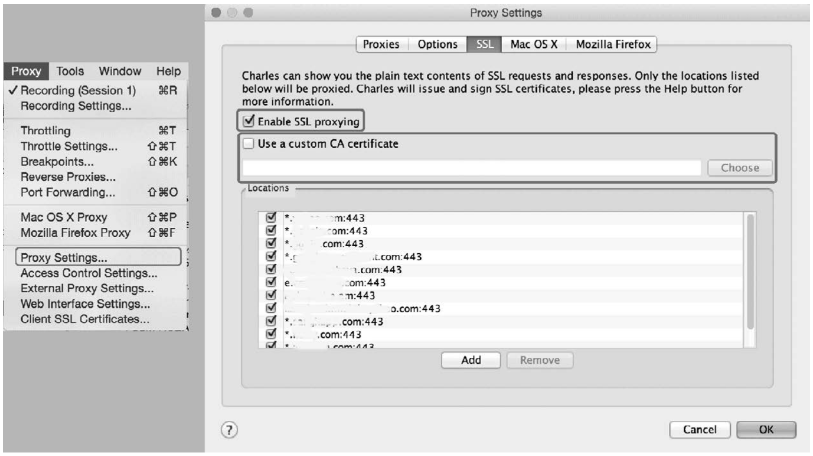


图 7-13：Charles：SSL 设置

> 仅在测试设备上使用网站的密钥。私钥和公钥都在公共领域。如果在个人设备上使用它们，你将会面临很大的风险——所有的通信都可能被监控。

- 发送自定义响应
  这是一个非常有用的功能，可以在不打扰生产服务器的前提下测试各种可能的情况。要测试性能，就发送大量的数据。要测试稳定性，就发送大量的数据以及无效的输入。

  你可以进入 Tools → Rewrite → Enable Rewrite → Add，如图 7-14 所示，并根据 URL 配置响应，如图 7-15 所示。

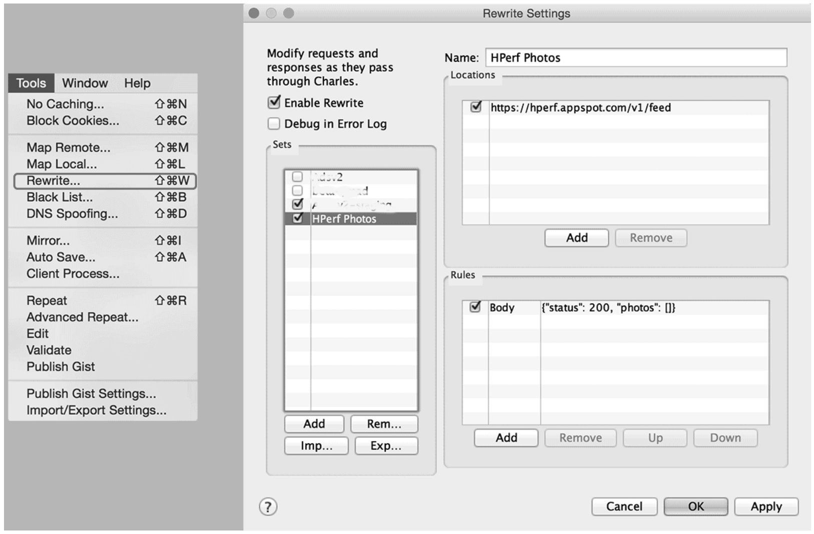


图 7-14：Charles：开启自定义响应

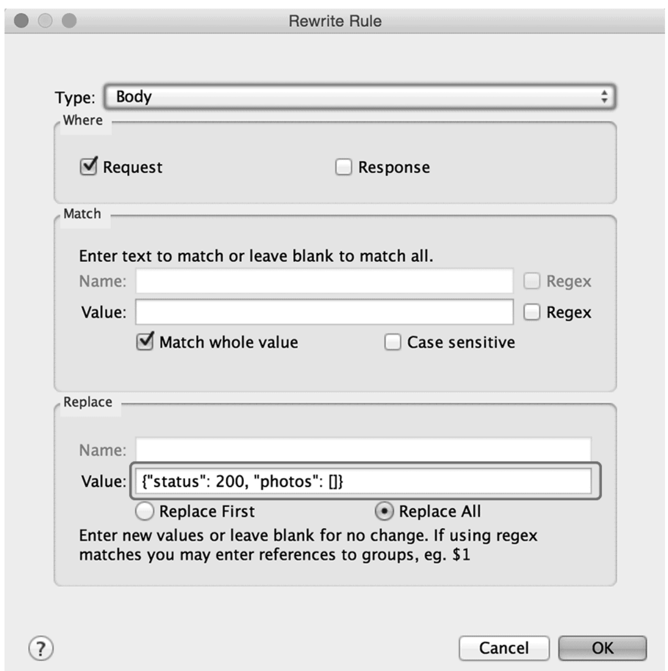


图 7-15：Charles：设置自定义响应

从开发、调试和测试的角度来看，请记住，会话日志可以被保存并分发给团队进行分析。

7.4 小结

了解与网络相关的重要指标以及用来测量网络的相关技术，将帮助你开发高性能的应用，让应用可以最大限度地减少任何资源的利用率，从而获得更好的用户体验。

这些做法通常需要进行权衡，因为对多个参数进行优化时，所需的条件可能会产生竞争关系，所以需要取得平衡。例如，为安全性使用 SSL 与内存利用率和执行速度的权衡；为了提高速度使用可缓存静态内容的 CDN 与为速度使用单个主机名的权衡，等等。

不要依赖网络状态，因为它随时都可以更改。你最好能确保自己的网络层适应于网络类型和状态。具体来说，对于媒体流，使用自适应多比特率 HLS。对于非媒体内容，执行批处理操作，并尝试尽可能多地预先下载。请务必提供适合设备的数据，因此需要考虑设备的性能、大小和外形尺寸。

工具应该提供不同的网络条件以及不同的数据响应来帮助你测试应用。使用这些工具来测试与网络、请求和响应相关的方方面面，从而加固应用。

第 8 章 数据共享

有时你会需要与其他应用共享数据，或访问设备上其他应用的共享数据。共享数据的场景包括以下几个。

- 与其他应用集成（例如，让用户使用 Facebook 的登录信息登录你的应用）。
- 发布一系列互补的应用，比如，Google 提供的应用（如 Gmail、Google Calendar、Google Hangouts 和 Google+）。
- 将用户数据从统一的应用移动到有多个特定用途的应用，检测其是否存在，并在需要时传递控制（例如，Facebook 应用分为 Messenger、Pages 和 Groups 应用，分别用于消息传递、页面管理和组管理）。
- 在可用的最佳查看器中打开文档（例如，在本地查看器中查看 PDF 文件，在 Photoshop Express 中编辑照片）。

用于共享数据的每种技术对可共享的数据具有特定的限制。例如，使用剪贴板会消耗大量的 RAM，文档的共享会使用设备存储（RAM 和设备存储必须在使用后清除）。类似地，使用深层链接会有数据序列化和解析的开销。

在本章中，我们将从性能角度讨论各种数据共享的选项，并明确一些特定情况下的最佳实践。

8.1 深层链接

在移动应用的上下文中，深层链接包括使用统一的资源标识符（uniform resourceidentifier，URI），其链接到移动应用内的特定位置，而不是简单地启动应用。

深层链接为应用之间的共享数据提供了解耦的方案。与访问网站时的 HTTP 网址类似，iOS 中的深层链接通过所谓的*自定义 URL scheme* 来提供。你可以配置自己的应用，让它

响应唯一的 scheme，操作系统会确保无论何时使用该 scheme，都由你的应用进行处理。应用可以响应任何数量的 scheme。

应用不能响应以下 URL scheme 的保留列表。

- `http`、`https`
  浏览网络的标准 scheme；由 Safari 处理。YouTube 链接是一个例外，如果已安装应用，则由 YouTube 打开（这是因为在创建自己的视频播放器之前，苹果已经与谷歌形成了合作伙伴关系）。

- `mailto`
  发送电子邮件的 scheme；由邮件应用处理，如 `mailto:email@domain.com`。

- `itms`、`itms-apps`
  用于将用户带入应用安装界面；由 App Store 应用处理。 这曾经是唯一可用的选项，直到 iOS 6 引入了 Store Kit（https://developer.apple.com/library/prerelease/ios/documentation/StoreKit/Reference/StoreKit_Collection/index.html）。

- `tel`
  用于呼叫电话号码；由电话应用处理，如 `tel://1234567890`。

- `app-settings`
  iOS 8 的新功能，此 scheme 带领用户去设置应用，并直接进入应用的设置部分。

选择唯一的 scheme

你选择的 URL scheme 在所有已安装的应用中必须是唯一的，否则会发生未定义的行为。[1]

你可以使用以下一种或多种方法来创建唯一的 scheme。

- 反向的 DNS 符号
  例如，如果你拥有的域名是 `yourdomain.com`，那就使用 `com.yourdomain.appname`。
- 包 ID
  由于所有提交到 App Store 的应用软件包的 ID 必须是唯一的，因此你可以使用此 ID。

- 应用 ID 加前缀
  在 App Store 中，每个应用都有一个唯一的数字 ID。你可以在它前面加几个字符，然后得到一个唯一的 ID。例如，如果应用的 ID 是 `1234567890`，你可以选择 scheme 为 `ios1234567890` 或 `app1234567890`。

无论使用何种选项，你唯一能期待的就是其他的应用不会使用相同的 scheme，因为这也许检测不出来。任何其他的恶作剧应用如果使用了相同的 scheme，而且未被发现，那么它可能会继续拦截本应属于你的应用的链接。

注 1：在 Android 系统上，如果多个应用响应一个 scheme，那操作系统会提示用户选择一个。类似功能将来有望引入 iOS 中。

除非选择一个通过争论可以获得所属权的 scheme，否则很难向苹果支持部门提出投诉。所以，要选择一个可以辩护的 scheme。例如，使用 com.yourdomain.appname 这样的 scheme 比 mail 或 song 更有说服力。

iOS 9 引入了通用链接，允许应用处理 http 或 https，同时可以验证域名的所有权。我们将在 13.1.1 节中对此进行讨论。

驱动深层链接有三个步骤。

(1) 检测 scheme 是否可以被处理。-[UIApplication canOpenURL:] 方法允许你在设备上安装的应用中检查是否至少有一个应用可以处理特定的 scheme。选择唯一的 scheme 有助于检测是否安装了特定的应用。

由于 scheme 的唯一性，自定义 URL scheme 可用于检测应用是否被安装。你可以使用版本后缀来检测是否安装了特定版本。
因此，应用可以支持多种 scheme。例如，Yelp（https://www.yelp.com/developers/documentation/v2/iphone）使用了三个 scheme——yelp5.3、yelp4 和 yelp，分别对应应用的 5.3.0 或更高版本、4.0.0 或更高版本，以及 2.0.0 或更高版本。因此，如果 canOpenURL: 为 yelp4 返回 YES，则表明设备上安装了 4.0.0 或更高版本，这可以帮助你选择不同的 URL，以获得更好的用户体验。
同样，你可以选择使用 com.yourdmain.appname+v1 和 com.yourdmain.appname+v2，分别对应应用的版本 1 和版本 2。

(2) 通过 URL 开启应用。一旦检测到应用的存在，下一步就是创建最终的 URL，并打开应用，使用 -[UIApplication openURL] 方法启动应用。

URL 格式

URL 的格式是没有标准的，甚至连约定也没有。应用使用了各种样式。URL 的一般格式是 scheme://host/path?query#fragment，其中的 path 可以使用正斜杠表示嵌套路径。

以下为使用的一些样式。

- 仅有路径的 URL
  这个想法是只使用路径来表示在处理数据时所需的所有细节。例如，fb://profile/{id}，Facebook 用它来显示用户的配置文件。
  它具有简单且易于理解的优点。

- 基于路径和查询的 URL

  这是更为通用和广泛使用的格式。例如，`yelp:///search?term=burritos`，Yelp 用它来搜索特定术语。

  `x-callback-url`（http://x-callback-url.com/）提出了使用 URL 格式 `{scheme}://{host}/{action}?{x-callback parame ters}&{action parameters}` 的标准，在该标准中，主机总是 `x-callback-url`。Tumblr、Google Maps、Google Chrome 和一些其他的应用（http://x-callback-url.com/apps/）都支持此格式。

  它的优点是创建和解析都较为简单。RFC 6874（https://tools.ietf.org/html/rfc6874）、RFC 3986（https://tools.ietf.org/html/rfc3986）和 RFC 1738（https://tools.ietf.org/html/rfc1738）这样的标准存在广义的 URL 格式。标准解析器的职责是解析更复杂的、可能有转义序列的查询字符串。

(3) 在目标应用中处理链接。当应用接收到 URL 时，`UIAppDelegate` 通过回调方法 `-[UIAppDelegate application:openURL:sourceApplication:annotation:]` 获取通知。解析传入的 URL、提取参数 / 值、处理，然后继续。

> 响应深层链接可能会将用户带往应用的其他部分，因此，你应该添加一个选项，让用户能够返回应用的前一个部分。一个较好的实现方案是使用有限的状态机。

例 8-1 显示了深层链接的示例——源和目标。

例 8-1　深层链接

```
//源应用——某个视图控制器
-(void)openTargetApp {
    NSURL *url = [NSURL URLWithString:
        @"com.yourdomain.app://x-callback-url/quote?ticker=GOOG\
            &start=2014-01-01&end=2014-12-31"]; ❶
    UIApplication *app = [UIApplication sharedApplication];
    if([app canOpenURL:url]) { ❷
        [app openURL:url]; ❸
    }
    //否则显示错误
}

//目标应用——应用委托
-(BOOL)application:(UIApplication *)application
    openURL:(NSURL *)url
    sourceApplication:(NSString *)sourceApplication
    annotation:(id)annotation { ❹

    NSString *host = url.host; ❺
    NSString *path = url.path; ❺
    NSDictionary *params = [self parseQuery:url.query]; ❺
    if([@"x-callback-url" isEqualToString:host]) { ❻
```

```
        if([@"quote" isEqualToString:path]) { ❻
            [self processQuoteUsingParameters:params]; ❼
        }
    }
    return YES;
}

-(NSDictionary *)parseQuery:(NSString *)query {
    NSMutableDictionary *dict = [NSMutableDictionary new];
    if(query) {
        //以'&'和'='作为分隔符解析
        NSArray *pairs = [query componentsSeparatedByString:@"&"]; ❽

        for(NSString *pair in pairs) {
            NSArray *kv = [pair componentsSeparatedByString:@"="];
            NSString *key = [kv.firstObject stringByRemovingPercentEncoding];
            NSString *value = [kv.lastObject stringByRemovingPercentEncoding];
        }

        [dict setObject:value forKey:key];
    }
    return [NSDictionary dictionaryWithDictionary:dict];
}

-(void)processQuoteUsingParameters:(NSDictionary *)params {
    NSString *ticker = [dict objectForKey:@"ticker"];
    NSDate *startDate = [dict objectForKey:@"start"];
    NSDate *endDate = [dict objectForKey:@"end"]; ❾
    //验证并处理
}
```

❶ 构造 URL。
❷ 检查应用是否被安装。
❸ 启动目标应用。
❹ 接收 URL 的目标应用中的委托回调。
❺ 从 URL 中提取必要的详细信息，包括主机、路径和查询。
❻ 处理 URL——此例为检查主机和路径。
❼ 在此示例中，处理引号。
❽ 解析查询字符串。此代码未优化，因为它会对字符串进行两次解析。
❾ 处理提取的值。不要忘记验证。

不论是访问共享数据，还是对外共享数据，深层链接可能是最常用的选项，同时，优化创建和解析的时间也很重要。以下列表涵盖了可以遵循的一些最佳实践，从而让应用实现最优性能。

- 最好使用较短的 URL，因为它们的构建速度和解析速度都比较快。
- 避免基于正则表达式的模式。
  如果使用 Button Deep Link SDK（http://www.usebutton.com/sdk/deep-links/），它将使用基于路径的 URL 和基于此的正则表达式。例如，路径模式 `{scheme}//say/:title/:message`（https://github.com/button/DeepLinkKit/blob/master/SampleApps/ReceiverDemo/

DPLReceiverAppDelegate.m）需要两个正则表达式——一个用于斜线分隔符，一个用于提取参数名称。

- 优先选择基于查询的 URL 进行标准解析。用基于字符的分隔符解析比使用正则表达式解析更快。
- 在你的 URL 中支持深层链接回调，以帮助用户完成意图。一个较好的方法是支持三个选项：`success`、`failure` 和 `cancel`。

  例如，如果一个照片编辑器应用能让用户将编辑的照片返回至照片应用中，那么必然会很受欢迎。另一个示例是，如果应用用于验证身份，那么提供一个能将用户带回到源应用的选项，并带上登录是否成功、是否被取消或失败的详细信息。

  `x-callback-url` 规范支持这些回调。
- URL 最好使用深层链接，以帮助用户定义一个需要多个应用协调的工作流。

  例如，用户可能想要完成以下操作。

  (1) 拍摄照片。
  (2) 编辑照片。
  (3) 将更新后的照片发送给家人和朋友。
  (4) 在社交媒体上分享更新后的照片。
  (5) 最后，返回照片应用来拍摄下一张照片。

  第一个应用可以定义深层链接的目标应用列表，以及在完成整个过程后所调用的最终的深层链接。
- 不要在 URL 中放置任何敏感数据。具体来说，不要使用任何身份验证令牌。这些令牌可能会被未知的应用劫持。
- 不要信任任何传入的数据。始终验证 URL。作为附加的措施，可以让应用在传递 URL 前对数据进行签名，并在处理前验证签名，这可能会是个不错的主意。但是，为了安全地进行，私钥必须保存在服务器上，如此一来，就必须要有网络连接。
- 使用 `sourceApplication` 来标识源。有一个应用白名单非常有用，你可以始终信任这些数据。`sourceApplication` 的使用与签名验证不正交。这可以是 URL 开始处理前的第一步。

8.2 剪贴板

官方文档对剪贴板的描述如下。

> 剪贴板是用于在应用之内或之间交换数据的安全且标准化的机制。许多操作取决于剪贴板，特别是复制—剪切—粘贴。……但你也可以在其他情况下使用剪贴板，例如，在应用之间共享数据时。[2]

可通过 `UIPasteBoard` 类使用剪贴板，该类可以访问共享存储库，写对象和读对象在共享存储库中进行数据交换。写对象也被称为剪贴板所有者，将数据存储在剪贴板实例上。读对

注 2：iOS Developer Library,“Pasteboard”(https://developer.apple.com/library/ios/documentation/General/Conceptual/Devpedia-CocoaApp/Pasteboard.html).

象访问剪贴板，将数据复制到其地址空间中。

剪贴板可以是公共的或私有的。每个剪贴板必须有唯一的名称。

剪贴板可以保存一个或多个实体，这些实体被称为剪贴板项目。每个项目可以有多个表述。

图 8-1 显示了一个具有两个项目的剪贴板，每个项目都有多种格式。

- 具有两种标准格式（RTF 和纯文本）内容的文本项。
- 具有两种标准格式（JPG 和 PNG）和一种私有格式（`com.yourdomain.app.type`）图片内容的图片项目，为特定应用所专用。

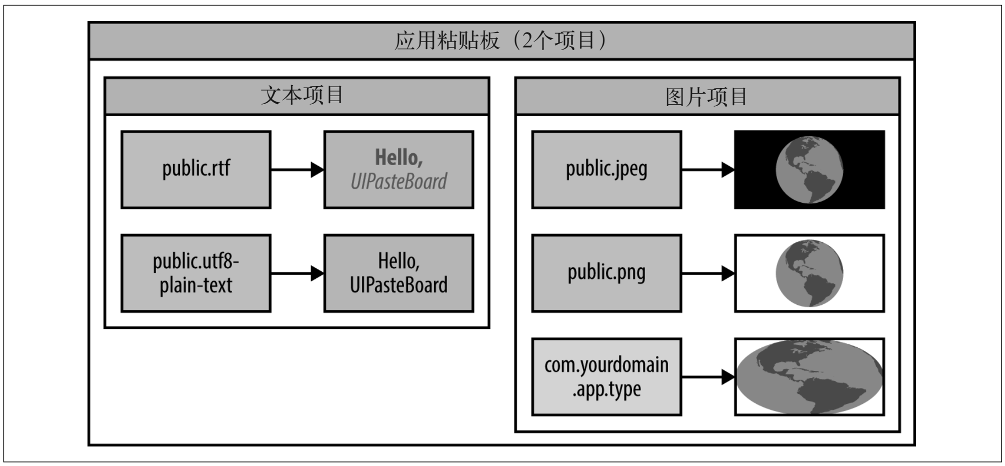


图 8-1：剪贴板项目表述

例 8-2 显示了共享数据的示例代码。

例 8-2　使用剪贴板共享数据

```
//共享至公共的剪贴板
-(void)shareToPublicRTFData:(NSData *)rtfData text:(NSString *)text {
  [[UIPasteboard generalPasteboard] setData:rtfData forPasteboardType:kUTTypeRTF]; ❶

  [[UIPasteboard generalPasteboard] setData:text forPasteboardType:kUTTypePlainText];
  [UIPasteboard generalPasteboard].string = text;
  [UIPasteboard generalPasteboard].strings = @[text];  ❷
}

//假设数据的UTI类型是"com.yourdomain.app.type"
-(void)shareToPublicCustomData:(NSData *)data {
  [[UIPasteboard generalPasteboard]
    setData:data
    forPasteboardType:@"com.yourdomain.app.type"];  ❸
}

//共享至自定义命名的剪贴板
-(void)sharePrivatelyCustomData:(NSData *)data {
  UIPasteboard *appPasteboard = [UIPasteboard
```

```
        pasteboardWithName:@"myApp"
        create:YES]; ❹

  [appPasteboard
    setData:data
    forPasteboardType:@"com.yourdomain.app.type"]; ❺
}

//从公共的剪贴板读取
-(NSArray *)readSharedStrings {
  return [UIPasteboard generalPasteboard].strings; ❻
}

//从命名的剪贴板读取
-(NSData *)readPrivateData {
  UIPasteboard *appPasteboard = [UIPasteboard
        pasteboardWithName:@"myApp"
        create:YES];

  return [appPasteboard
      dataForPasteboardType:@"com.yourdomain.app.type"]; ❼
}
```

❶ 设置已知类型 kUTTypeRTF 的二进制数据。
❷ 可以为 kUTTypePlainText 类型设置纯字符串，也可以使用 string 属性。对 NSString 对象的数组而言，使用 strings 属性也是可行的。
❸ 为自定义类型设置二进制数据。
❹ 获取具有给定名称的剪贴板。若不存在，新建一个。
❺ 设置自定义剪贴板的数据。
❻ 检索存储在 strings 属性中的字符串数组。
❼ 在自定义剪贴板（非公开）中检索自定义类型的数据。

与深层链接相比，剪贴板具有以下优点。

- 它具有支持复杂数据（如图像）的能力。
- 它支持在多种形式中表示数据，这些形式可以基于目标应用的功能来选择。例如，消息应用可以使用纯文本格式，邮件应用可以使用来自同一剪贴板项目的富文本格式。
- 即使应用关闭后，剪贴板内容仍然会保留。

然而，与深层链接相比，使用剪贴板的明显缺点是共享数据的格式不是任何标准格式。因此，如果不能定义两个应用之间的数据契约，那么剪贴板将不能用于通用共享。

如例 8-2 所示，使用多个选项可以共享纯文本数据；有时，选择使用哪一种格式会令人感到困惑。

与深层链接不同，剪贴板不能用于检测是否安装了目标应用。此信息有助于提供更好的用户体验，例如，如果尚未安装应用，那么可以提示用户进行安装。

此外，与深层链接不同，任何应用都可以访问剪贴板，因此它会带来深层的安全问题。

使用剪贴板时，你应该遵循以下的最佳实践。

- 剪贴板本质上是由剪贴板服务进行调解的进程间通信。IPC 的所有安全规则都适用（例如，不发送任何安全数据、不信任任何传入数据）。
- 因为不能控制哪个应用会访问剪贴板，所以使用时总是不安全的，除非数据被加密。
- 不要在剪贴板中使用大量数据。虽然剪贴板支持交换图像以及多种格式，但请记住，每个条目不仅消耗内存，也需要额外的时间来读写。
- 当应用将使用 `UIApplicationDidEnterBackgroundNotification` 通知或 `UIApplicationWillResignActiveNotification` 通知进入后台时，清除剪贴板。更好的做法是，你可以实现 `UIApplicationDelegate` 相应的回调方法。

  通过将 `items` 设置为 `nil`，你可以清除剪贴板，如下所示：

```
myPasteboard.items = nil;
```

- 为了防止任何类型的复制 / 粘贴，继承 `UITextView`，并在 `canPerformAction` 的 `copy:` 动作中返回 `NO`。[3]

8.3　共享内容

我们前面探索的两个选项——自定义 URL scheme 和剪贴板——完全是机器驱动的，不受人为控制。终端用户不能选择要进入哪个应用或数据将用于哪个应用。

为了填补这个空白，iOS 提供了几个选项用于与特定应用共享文档。源应用生成要共享的数据，用户选择将要使用数据的应用。图 8-2 显示了使用文档共享选项的 WhatsApp 和 Photos 应用。可用的应用列表取决于所选的选项，我们将在下面进行探讨。

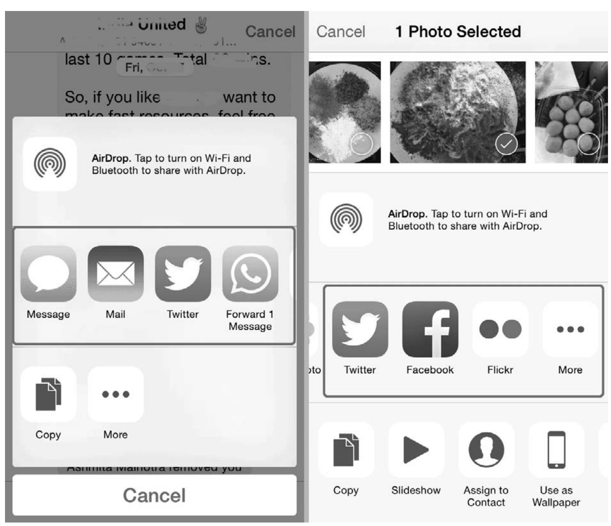


图 8-2：WhatsApp 和 Photos 应用分别分享消息和照片

注 3：Stack Overflow,“How to Disable Copy, Cut, Select, Select All in UITextView”(http://stackoverflow.com/a/1429320).

8.3.1　文档交互

自 iOS 3.2 起就已经可用的 `UIDocumentInteractionController` 类允许应用利用设备上的其他应用打开文档。它还支持预览、打印、邮寄和复印文档。

> `UIDocumentInteractionController` 不是 `UIViewController` 的子类。你必须配置一个视图控制器来预览文档。

控制器的使用涉及两个方面：发布者和用户。

作为发布者，你的应用要发布即将被查看的文档。控制器负责加载目标应用，让内容对用户应用可用，最后还要让用户回到主应用。本节的“发布者”中介绍了发布者方面的典型代码。

作为用户，你的应用的职责包括处理文档（并呈现结果），还负责执行一些清理工作，如本节的“消费者”所示。

图 8-3 说明了文档共享期间所要经过的步骤。

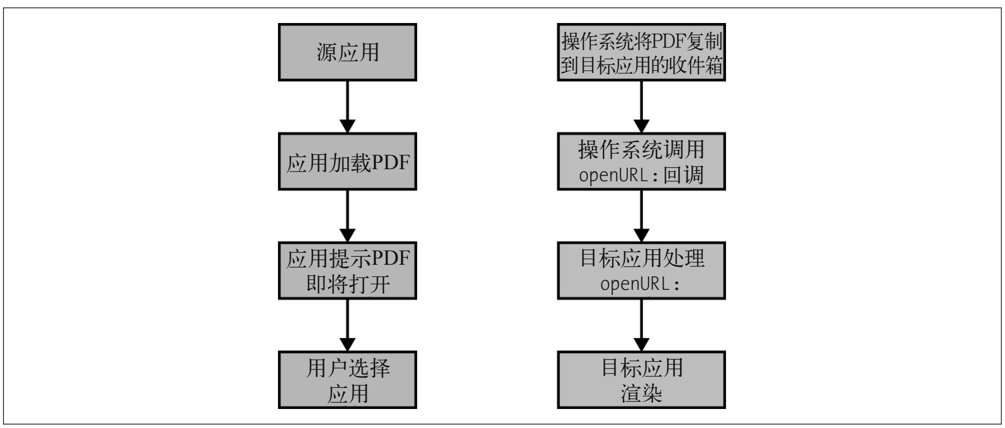


图 8-3：完整的文档共享过程

1. 发布者

作为发布者，应用可以预览或打开文档。`UIDocumentInteractionController` 通过 `UIDocumentInteractionController Delegate` 作为主应用的委托，指定父视图控制器显示预览窗口。视图控制器还需要文档的路径，以及与文档类型相关联的统一类型标识符。

统一类型标识符

统一类型标识符（uniform type identifier，UTI）是用来唯一标识某一项目类型的文本字符串。内置的 UTI 用来标识公共系统对象。例如，文档的 `public.document`、JPEG 图像的 `public.jpeg` 和纯文本的 `public.plain-text`。

有一个规定允许第三方开发人员添加自己的 UTI，用于特定应用或专有用途。例如，用于 PDF 文档的 `com.adobe.pdf` 和用于 Apple Keynote 文档的 `com.apple.keynote.key`。

委托获取一个回调以确定是否可以执行特定操作。这些操作默认情况下包括 `copy:`、`print:` 和 `saveToCameraRoll:`。在预览 / 打开的 UI 即将呈现之前以及呈现之后，它都会获得相应的回调。

例 8-3 显示了对共享文档进行预览和打开的示例代码。

例 8-3　文档交互——发布者

```
#import <MobileCoreServices/MobileCoreServices.h> ❶

@interface HPDocumentViewerViewController
    <UIDocumentInteractionControllerDelegate> ❷

@property (nonatomic, strong) UIDocumentInteractionController *docController;

@end

@implementation HPDocumentViewerViewController

-(UIViewController *)documentInteractionControllerViewControllerForPreview:
    (UIDocumentInteractionController *) controller { ❸
    return self;
}

-(NSURL *)fileInDocsDirectory:(NSString *)filename {
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
        NSUserDomainMask, YES);
    NSString *docsDir = [paths firstObject];
    NSString *fullPath = [docsDir stringByAppendingPathComponent:filename];

    return [NSURL fileURLWithPath:fullPath];
}

-(void)configureDIControllerWithURL:(NSURL *)url
    uti:(NSString *)uti { ❹

    UIDocumentInteractionController controller = [UIDocumentInteractionController
            interactionControllerWithURL:url]; ❺
        controller.delegate = self;
        controller.UTI = uti; ❻
        self.docController = controller; ❼
}
```

```
-(IBAction)previewDocument {
    NSURL *fileURL = [self fileInDocsDirectory:@"sample.pdf"]; ❽

    if(fileURL) {
        [self configureDIControllerWithURL:fileURL uti:kUTTypePDF];
        [self.docController presentPreviewAnimated:YES]; ❾
    }
}

-(IBAction)openDocument {
    NSURL *fileURL = [self fileInDocsDirectory:@"sample.pdf"];

    if(fileURL) {
        [self configureDIControllerWithURL:fileURL uti:kUTTypePDF];

        [self.docController presentOpenInMenuFromRect:self.view.frame
            inView:self.view animated:YES]; ❿
    }
}
@end
```

❶ UIDocumentInteractionController 类、相关的类型以及常量定义在 MobileCoreServices 中。

❷ 控制器需要 UIDocumentInteractionControllerDelegate。让视图控制器实现协议。

❸ 虽然所有的方法都是可选的，但你必须实现方法 documentInteractionControllerViewControllerForPreview:，因为该方法提供了将要展示子视图控制器的 UIViewController。

❹ 辅助方法使用内容 URL 和 UTI 类型配置控制器。

❺ 获取 URL 指向的控制器的引用。

❻ 指定委托和 UTI 类型。

❼ 设置对控制器的强引用，确保控制器不会过早地被释放。

❽ 请注意，UIDocumentInteractionController 对象引用的 URL 必须是操作系统可访问的。如果需要，那么下载文件的内容，并使用本地（文件）URL 进行引用。

❾ 使用 presentPreviewAnimated: 预览文档。如图 8-4 所示，在应用中展示有动画效果的视图控制器。

❿ 使用 presentOpenInMenuFromRect:inView:animated: 显示“在……中打开”的菜单，并让用户选择某一应用打开文档。图 8-5 显示了应用中的菜单。

> UIDocumentInteractionController 需要一个 NSURL 来读取内容，它必须指向一个使用文件 scheme 的本地文件。任何其他 scheme 将引发异常，最终有可能导致应用崩溃。

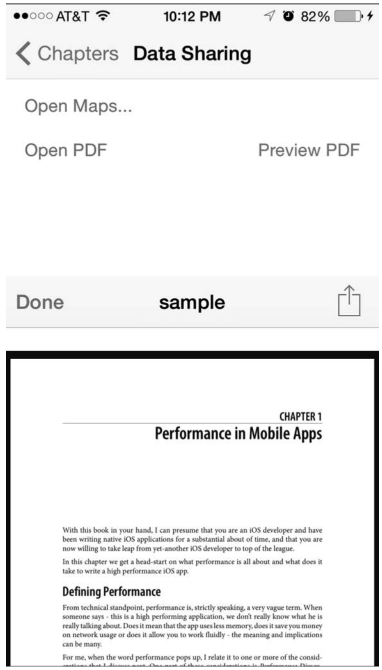


图 8-4：预览 PDF 文档（显示的视图控制器）

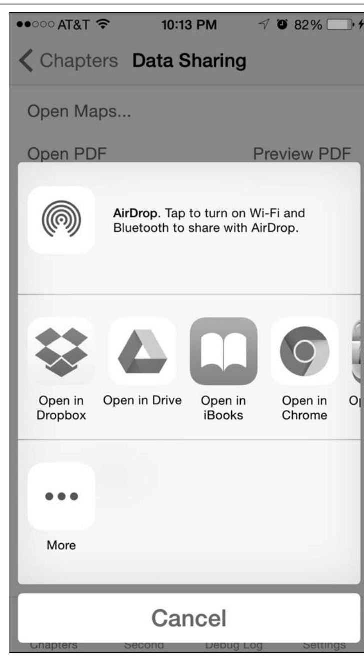


图 8-5：PDF 文档中的“在……中打开”菜单

2. 消费者

作为文档的消费者需要两个基本步骤：注册应用支持的文件类型，然后处理文档内容。你可以支持一个或多个 iOS 系统定义的类型，也可以注册新类型，注册的新类型有助于在同一公司或其他公司的一系列应用之间进行共享。

要想注册应用支持的文件类型，你必须在应用的 `Info.plist` 中的文档类型部分配置以下详细信息。

- 名称
  你想提供的人类可读的名称。
- 类型
  标准的统一类型标识符[4]之一（如 `com.adobe.pdf`）或自定义 UTI。
- 图标
  如果与应用图标不同，则需要一个与文档相关联的图标。
- 属性
  作为可选项，你可以配置其他文档类型属性。

图 8-6 显示了应用的文档类型部分。请注意，类型已经被设置为 `com.adobe.pdf`，这是一种预定义类型。随意选择一个自定义名称，以便在各组应用中共享自定义类型。必须将相同的值用作 `UIDocumentInteractionController` 对象的 `UTI` 属性。

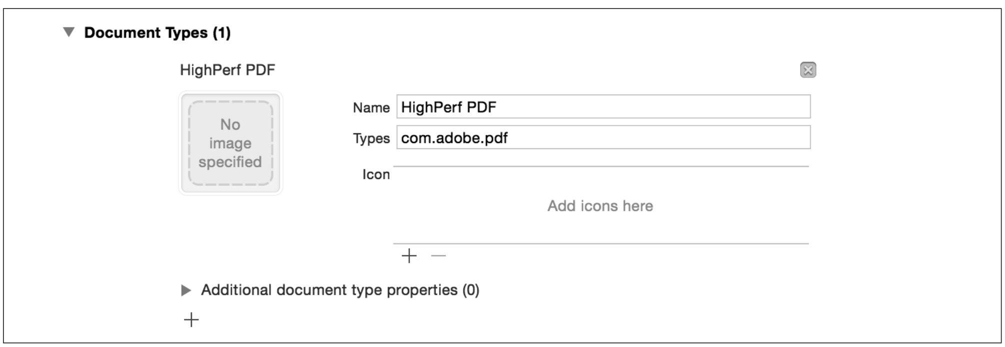


图 8-6：配置文档类型以处理 PDF 文档

使用此设置，如果在 Safari 中打开 URL：https://bitcoin.org/bitcoin.pdf，然后点击“在……中打开”菜单，那么它会显示出我们的应用（见图 8-7）。

注 4：有关完整列表，请参阅 iOS 开发者库（https://developer.apple.com/library/ios/documentation/Miscellaneous/Reference/UTIRef/Articles/System-DeclaredUniformTypeIdentifiers.html#//apple_ref/doc/uid/TP40009259-SW1）。

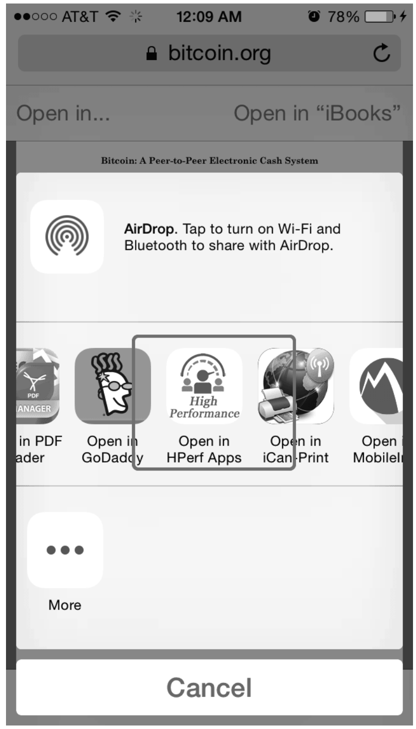


图 8-7：查看 PDF 文档时，在 Safari 中的“在……中打开”菜单

现在，我们必须处理应用的委托回调 `openURL:sourceApplication:annotation:`。如图 8-3 所示，共享文档被复制到应用的*收件箱*文件夹中。传递给回调的 `url` 是一个指向文件的文件 URL。如果用户对同一应用多次共享同一文档，那么操作系统会创建文档的多个副本，且每次都有新的可用 URL。例 8-4 显示了处理文档的典型代码。

例 8-4　文档交互——消费者

```
@implement HPAppDelegate

-(BOOL)application:(UIApplication *)application
    openURL:(NSURL *)url sourceApplication:(NSString *)src
    annotation:(id)annotation {

    DDLogDebug("%s src: %@, url: %@", __PRETTY_FUNCTION__, src, url);
        return YES;

}

@end
```

如果查看图 8-8 中应用委托的日志，你会注意到，即使在同一个应用中打开同一文档，每次的 `url` 也是不同的。



图 8-8：应用委托回调的调试日志

使用 `UIDocumentInteractionController` 共享文档时需要遵循的最佳实践，与我们之前讨论共享选项时的最佳实践类似。此外，我们更应该关心另一个问题。

如上所述，使用 `UIDocumentInteractionController` 会导致文档被复制到应用的收件箱文件夹中。因此，应用有责任删除文件并清理文件夹。生成的文件由用户应用所有。完成后，不要忘记删除文件。

8.3.2 活动

UTI 在过去可以很好地工作。然而，云服务和社交媒体的兴起使远程实体优于本地文件。因此，UTI 和远程 URL 之间的关系变得紧张。

`UIActivityViewController` 提供了一个统一的服务接口，以便共享和执行与应用中数据有关的操作。它在 iOS 6 中被引入。

使用 `UIActivityViewController` 比使用 `UIDocumentInteractionController` 更容易、更灵活。`UIDocumentInteractionController` 只允许文件 URL，但使用 `UIActivityViewController` 可以共享以下一种或多种类型。

- `NSString`

  任何字符串都可以共享。

- `NSAttributedString`
  可以共享格式化文本或富文本。
- `NSURL`
  任何 URL 都可以共享。根据目标应用的不同而使用不同的 URL。邮件或消息应用可以选择按原样共享 URL，而阅读器或云服务应用可能会尝试获取内容并进行处理。
- `UIImage`
  如果提供了图像，那么图片也可以保存到相机胶卷，分配给联系人或打印。
- `ALAsset`
  这展示由照片应用管理的照片或视频，可以与目标应用共享。
- `UIActivityItemSource`
  符合此协议的对象可以共享。这有助于创建可以跨应用共享的自定义对象。

> 要想将 `UIImage` 保存到相机胶卷或使用 `ALAsset`，应用需要获得访问照片的权限。如果应用从未询问过，那么系统会提示用户授予权限。如果用户之前已经拒绝或授予访问权限，则不会出现权限提示。

有两种类型的活动：操作和共享（见图 8-9）。共享从第三方应用（如 Facebook、Twitter、Vimeo 等）启动 UI，而操作主要涉及内置的应用（照片、打印机、剪贴板 / 复制、Safari、联系人等）。此外，还有 AirDrop 支持图像。苹果公司的开发者网站（https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIActivity_Class/index.html#//apple_ref/doc/constant_group/Built_in_Activity_Types）上提供了内置操作类型的完整列表。



图 8-9：使用 `UIActivityViewController`——操作和共享活动

例 8-5 显示了在应用中启用活动视图控制器的典型代码。你可以选择排除一个或多个活动。此外，通过实现自定义 UIActivity，你可以让用户将内容共享到同一个应用，UIActivity 是一个必须被子类化的抽象类。

例 8-5　使用 UIActivityViewController 共享数据

```
-(void)shareSomeContent {
    NSString *text = @"Text to share";
    NSURL *url = [NSURL URLWithString:@"http://github.com"];
    UIImage *image = [UIImage imageNamed:"blah"]; ❶

    NSArray *items = @[text, url, image];
    UIActivityViewController *ctrl = [[UIActivityViewController alloc]
        initWithActivityItems:items
        applicationActivities:nil]; ❷

    ctrl.excludedActivityTypes = @[ UIActivityTypePostToFacebook ]; ❸
    [self presentViewController:ctrl animated:YES completed:nil]; ❹
}
```

❶ 要共享的几个项目——字符串、URL 和图片。
❷ 使用活动项目实例化 UIActivityViewController。此示例没有配置任何 applicationActivities。如果需要，它们必须是 UIActivity 的子类对象。
❸ 排除不允许的活动类型——此处已排除在 Facebook 上发帖子。
❹ 最后，呈现视图控制器，使用模式（如这里演示的）或使用 navigationController。

就与其他应用共享内容而言，活动非常灵活、可扩展，而且功能强大。但性能和安全问题是之前讨论的其他数据共享方式中存在的问题的集合。

共享钥匙串

共享钥匙串是在应用间安全共享数据的另一种选择。只有属于相同群组 ID，且使用相同证书签名的应用才能共享数据。

在你的所有应用中实现单点登录的唯一方法就是使用共享钥匙串。

要实现在同一发布者（相同的签名证书）的应用之间共享数据，这是唯一的方式，且不需要从用户正在使用的应用调用其他应用。

因为数据是加密的，所以它应该是存储安全信息的地方，如凭证、信用卡号（虽然没有 CVV）等。避免在大量通用、非安全数据中刷新，因为这样的访问会比对未加密数据的访问慢。

8.4　iOS 8扩展

iOS 8 引入了四个新选项，用于在应用之间共享内容。这四个选项被用在应用扩展（在第 6 章中简要介绍过）的更广泛类别之中。

如果打开项目并点击加号图标（+）来添加目标（见图 8-10），你应该可以在 iOS 下找到一

个新的应用扩展条目，该条目会提供如图 8-11 所示的选项。

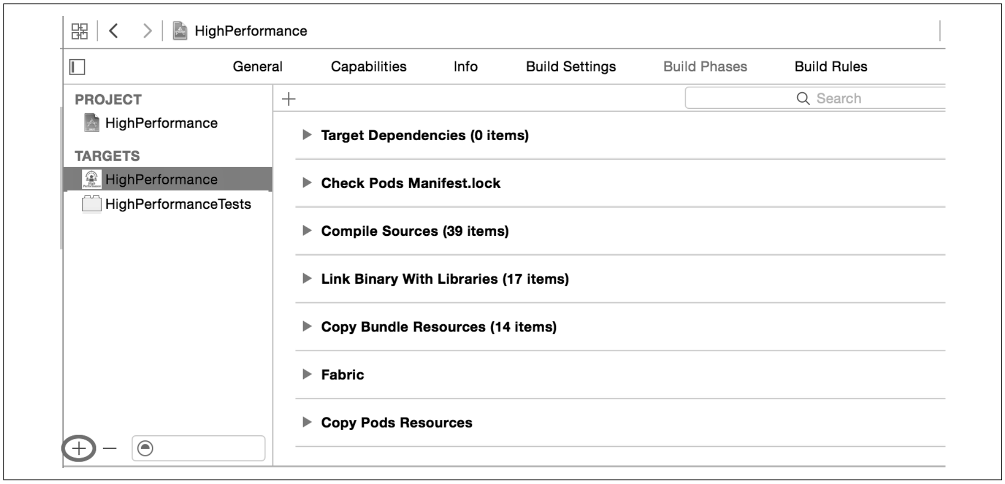


图 8-10：Xcode → Project → Add Target

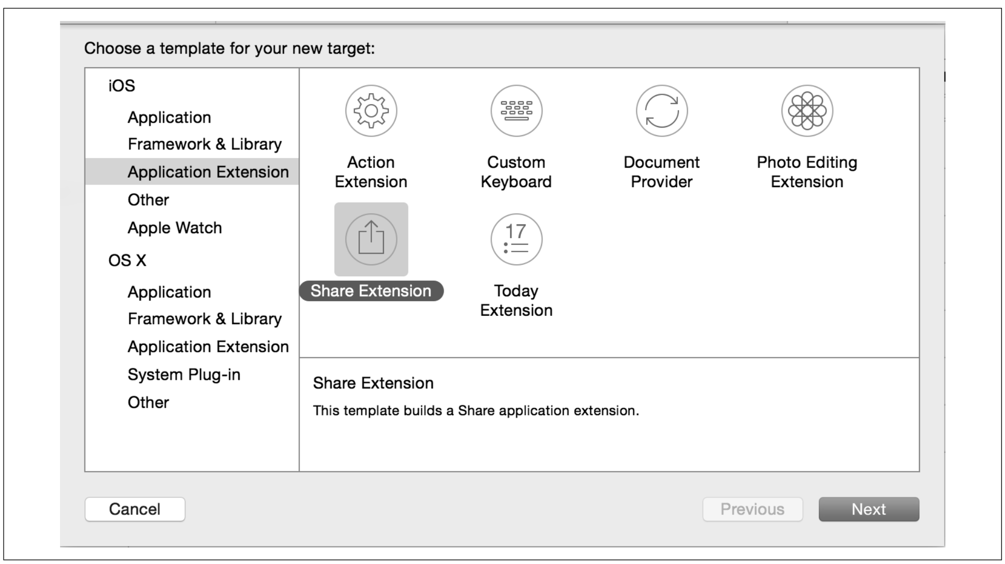


图 8-11：选择 iOS → Application Extension

在所有的选项中，只有以下几项是用来实现数据共享的：

- 操作扩展
- 分享扩展
- 文档提供者扩展
- 应用群组

从实现的角度看，我们对它们并不陌生。在上一节中，我们已经讨论过通过动作进行分享、

使用 UIActivityViewController 实现共享，以及使用 UIDocumentInteractionController 实现文档共享。

与之前相比，本节介绍的较为新鲜的内容是整体的探索、实施的难易程度，以及为终端用户提供的选项。

在探索这些共享数据的扩展之前，我们先探讨在 iOS 8 中添加的一些新类。

- NSExtensionContext（http://apple.co/1HejEzB）
  表示调用扩展时的主应用的上下文。它提供了输入项的一个数组（例如，共享给应用的数据）。
- NSExtensionItem（http://apple.co/1BFpuls）
  表示输入项数组中的某一项。NSExtensionItem 对象是一个不可变集合，集合中的值代表了项目的不同方面，可通过 attachments 属性获得。
- NSItemProvider（http://apple.co/1ECgC4H）
  表示可在 NSExtensionItem 对象的 attachments 属性中找到的数据对象，如文本，图像和 URL。使用 hasItemConformingToTypeIdentifier: 方法检查其表示的 UTI 类型。要检索相应类型的数据，请使用方法 loadItemForTypeIdentifier:options:completionHandler:。回调方法可以在任何线程中被调用——更改 UI 时，不要忘记将上下文切换到主线程。

8.4.1 配置操作扩展和共享扩展

对于操作和共享扩展而言，除了特定项目，二者对所有模板都是通用的。

- 元数据（Info.plist）
  最重要的两个条目是包显示名称和 NSExtensionItem，前者指的是在条目旁边显示的名称，后者提供了何时在列表中显示此操作的元数据。Xcode 将 NSExtension → NSExtensionAttributes → NSExtensionActivationRule 的值设置为 TRUEPREDICATE（类型为 String），实质上表示该操作始终可用。可以将其更改为 Dictionary 类型，并使用表 8-1 中列出的键来提供更细粒度的控制。

表8-1：应用扩展键

键	描述
NSExtensionActivationSupportsText	应用支持文本（NSString 或 NSAttributedString 类型的值）
NSExtensionActivationSupportsFileWithMaxCount	应用支持任何文件的处理（使用文件 scheme 的 NSURL）
NSExtensionActivationSupportsWebURLWithMaxCount	应用支持网页 URL（使用 http 或 https scheme 的 NSURL）
NSExtensionActivationSupportsWebPageWithMaxCount	应用支持网页
NSExtensionActivationSupportsImageWithMaxCount	应用支持图像（UIImage 值）
NSExtensionActivationSupportsMovieWithMaxCount	应用支持视频
NSExtensionActivationSupportsAttachmentsWithMinCount	扩展要激活的最小附件数（默认为 1）
NSExtensionActivationSupportsAttachmentsWithMaxCount	扩展要激活的最大附件数（默认为 long max 值）

正值表示可以为特定类型共享的条目的最大值。例如，NSExtensionActivationSupportsImageWithMaxCount 的值为 2，这表示最多可以共享两个图像。缺少键或值为 0 则意味着扩展不支持此特定类型。要声明一个更复杂的定义，你可以使用 NSPredicate-compilable 结构。请参阅“应用扩展编程指南”（http://apple.co/1LDgW9X）中的“为共享或操作扩展声明其支持的数据类型”部分。

- 目标产品名称
  当创建新扩展时，使用产品名称字段中提供的名称创建新目标。

8.4.2 操作扩展

在使用 UIActivityViewController 时，操作扩展允许你将视图控制器添加到操作部分。iOS 7 捆绑了来自其他应用的预定义操作列表，但是没有办法增加更多的，iOS 8 更改了这一点。

当你创建操作扩展时，Xcode 将创建以下的附加内容。

- 故事板主界面
  当用户选择该操作时，故事板 UI 即将显示出来。
- ActionViewController 类
  支持故事板的视图控制器类。

  作为一个视图控制器，它有和其他控制器一样的生命周期（回忆一下之前的图 6.1）。

例 8-6 展示了渲染源应用共享图像的典型代码。

例 8-6 操作——从共享数据渲染图片

```
- (void)viewDidLoad {
    [super viewDidLoad];
        BOOL imageFound = NO;
        for(NSExtensionItem *item in self.extensionContext.inputItems) { ❶
                for(NSItemProvider *itemProvider in item.attachments) { ❷
            if([itemProvider
                    hasItemConformingToTypeIdentifier:(NSString *)kUTTypeImage]) { ❸
                               [self processItem:itemProvider];
                               imageFound = YES;
                               break;
                        }
        }
                if(imageFound) {
                        break;
                }
        }
}

-(void)processItem:(NSItemProvider *)itemProvider {
    UIImageView __weak *imageView = self.imageView;
        [itemProvider loadItemForTypeIdentifier:(NSString *)kUTTypeImage
        options:nil
        completionHandler:^(UIImage *image, NSError *error) { ❹
```

```
                if(image) {
                        [[NSOperationQueue mainQueue] addOperationWithBlock:^{
                                [imageView setImage:image];
                        }]; ❺
                    }
            }];
    }
```

❶ 扫描所有的扩展项。
❷ 对于每个项目而言，扫描所有的附件。
❸ 检查附件是否为图像类型。
❹ 如果是，请检索内容。
❺ 因为检索回调可以在非主线程上被调用，所以切换上下文，以便使用 `UIImage` 内容更新 `UIImageView`。

8.4.3 共享扩展

共享扩展与共享活动略有不同，前者是系统提供的 UI，无法由接收应用自定义。

> 用户可以在系统提供的 UI 中访问共享扩展。在 iOS 中，用户点击共享按钮，然后从显示的活动视图控制器的共享区域中选择共享扩展。[5]

创建共享扩展时，Xcode 将创建以下的附加内容。

- 故事板主界面
  至少在 iOS 8.2 之前，这没有什么意义。在未来，苹果公司可能会允许应用提供自定义 UI。
- `ShareViewController` 类
  这是在 iOS 8 中引入的 `SLComposeServiceViewController` 的子类。虽然它是一个视图控制器，但控制器配置的 UI 完全被忽略了。

这个类提供了以下生命周期事件的挂钩。

- 内容验证
  调用的第一个方法是 `isContentValid`。使用 `NSExtensionContext` 验证传入的值（见例 8-6），如果数据有效则返回 `YES`，如果无效就返回 `NO`。无论值是什么，该活动将始终被显示，但如果内容无效，Post 按钮将被禁用（见图 8-12）。
- `viewDidLoad`
  该方法在验证初始内容及加载视图后被调用。如果需要的话，使用 `textView` 属性访问 `UITextView` 编辑器，从而对文本进行更改。
- 获取配置项
  视图加载后，`configurationItems` 方法会被调用来检索配置项。它返回一个值为 0 的数组或多个 `SLComposeSheetConfigurationItem`（子类）条目的数组。

注 5：iOS Developer Library，“Share”(https://developer.apple.com/library/ios/documentation/General/Conceptual/ExtensibilityPG/Share.html).

SLComposeSheetConfigurationItem 对象使用标准的 UITableView，以提供要显示在共享编辑器下面的标题和值（见图 8-12）。

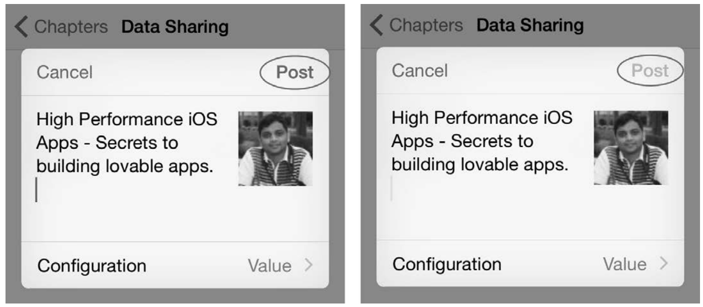


图 8-12：有效内容（左）和无效内容（右）的共享活动

SLComposeSheetConfigurationItem 对象还提供了一个 tapHandler，用户点击配置项时会调用后者。tapHandler 可以压入一个视图控制器，用来显示对值进行修改的一些选项。例如，Facebook 允许更改图像的相册、位置和隐私值（见图 8-13）。

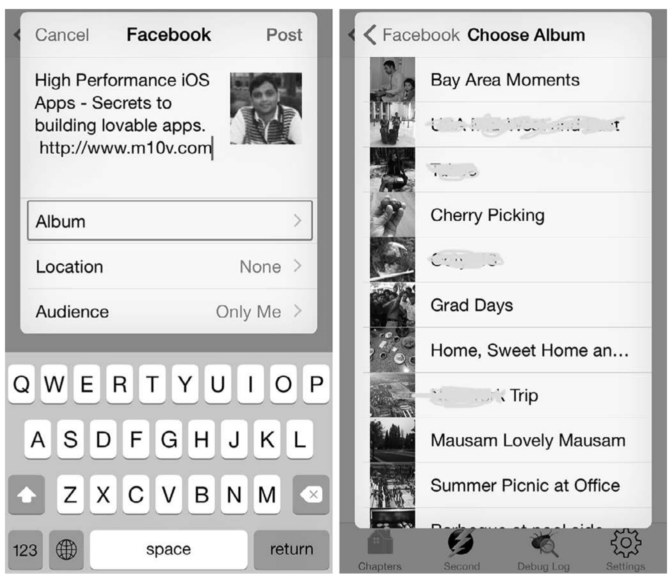


图 8-13：Facebook 共享——活动（左）和相册配置（右）

- 视图控制器生命周期方法
  随后调用 viewWillAppear 和 viewDidAppear 方法。
- 内容验证
  在显示视图后立即执行内容验证的最后一段。

- 内容变化的验证

  每当用户更改编辑器中的内容时，调用 `isContentValid`。扩展可以调用 `validateContent` 方法来触发重新验证，或调用 `reloadConfigurationItems` 方法来重新加载配置项。它还可以实现 `charactersRemaining` 方法，以返回一个非负值，表示剩余的字符数（图 8-14 展示了一个显示该值的 Twitter 共享）。



图 8-14：Twitter 共享，展示剩余的字符数

- 取消通知

  如果用户点击取消按钮，`didSelectCancel` 会被调用。

- 发布通知

  如果用户点击发布按钮，`didSelectPost` 会被调用。

> 使用共享扩展时，请注意以下几点。
>
> - 所有方法都是在用户选择活动后被调用的。
> - 对于取消和发布通知，调用 `NSExtensionItem` 的 `completeRequestReturningItems:completionHandler:` 方法，以表示活动的交互已经完成了。否则，源应用会处于不可用状态。作为一种最佳实践，应该根据 iOS 8 或更高版本构建应用。当想在应用中使用活动时，检测应用是否在 iOS 7 或更早版本上执行。如果是，请使用自定义活动视图控制器。如果不是，让操作系统为你选择活动。

8.4.4 文档提供者扩展

文档提供者是文档交互 API 的 iOS 8 扩展版本。要想读取共享文档的内容，请使用 `UIDocumentPickerViewController`。要呈现一个 UI 来共享文档，则应该子类化 `UIDocumentPickerViewController`。

使用文档提供者需要 iCloud 授权。进入 Project → App → iCloud，选择 iCloud Documents，如图 8-15 所示。

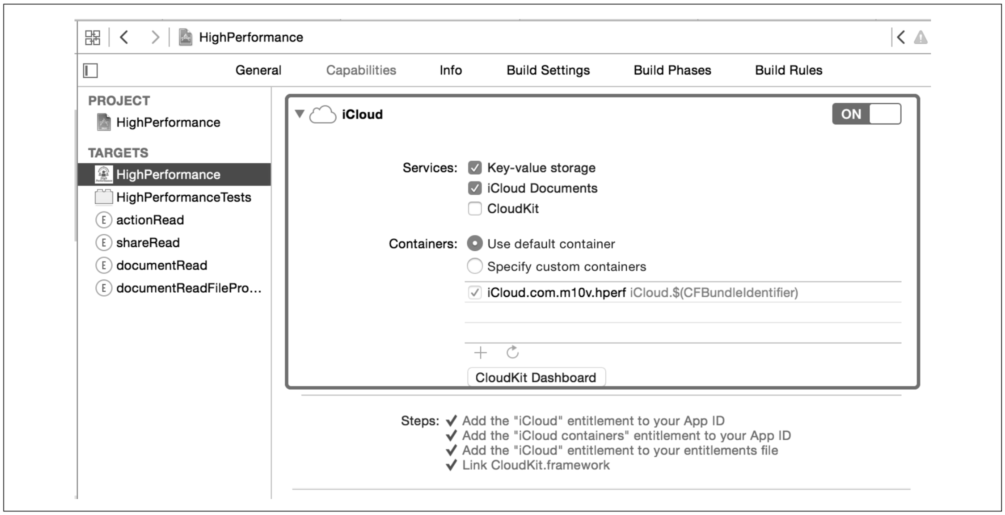


图 8-15：应用清单——iCloud 授权

1. 打开/导入文档

`UIDocumentPickerViewController`（通常被称为文档选择器）提供了一个钩子，以便与安装在设备上的其他文档提供者进行交互。文档选择器可以在打开 / 导入模式或导出模式下工作。

该方法类似于文档交互提供者的方法，只是前者是从“在……中打开”的菜单进入的。因此，用户无需立即进入提供文档的应用（如浏览器、Google Drive、Dropbox 等），而是直接在相关应用中导入文档并继续操作。

图 8-16 显示了使用文档交互方式编辑文档和使用文档选择器编辑文档时，每个步骤的差别。

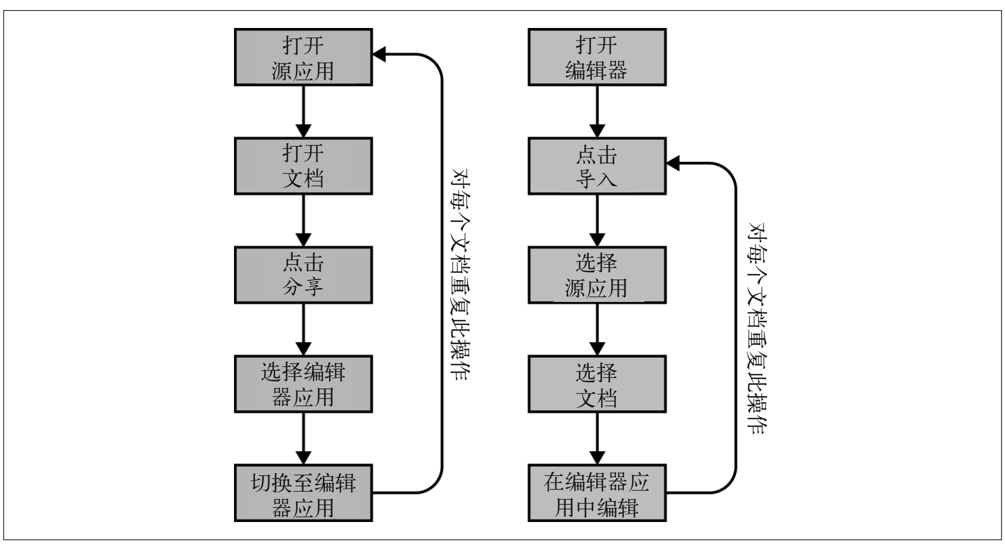


图 8-16：使用活动或文档交互（左）和使用文档选择器（右）编辑文档

UIDocumentPickerViewController 对象需要配置以下项目。

- 文档类型
  编辑器应用可以支持的 UTI 类型。
- 模式
  必须被配置为 Open 或 Import。
- 委托
  当用户选择文档时，UIDocumentPickerDelegate 类型的委托会发生响应。在用户取消选择时，它也可能（可选）响应。

例 8-7 显示了从其他文档提供者打开 / 导入文档的典型代码。

例 8-7　文档提供者——打开 / 导入

```
@interface HPDocumentEditorViewController
    : UIViewController <UIDocumentPickerDelegate> ❶
@end

@implementation HPDocumentEditorViewController
-(IBAction)openButtonWasClicked:(id)sender {
    NSArray *types = @[
        (NSString *)kUTTypeImage
        ]; ❷

        UIDocumentPickerViewController *dpvc =
        [[UIDocumentPickerViewController alloc]
                        initWithDocumentTypes:types
            inMode:UIDocumentPickerModeImport]; ❸
        dpvc.delegate = self; ❹
        [self.navigationController presentViewController:dpvc
        animated:YES completion:nil]; ❺
}

-(void)documentPicker:(UIDocumentPickerViewController *)controller
    didPickDocumentAtURL:(NSURL *)url { ❻
    NSData *data = [NSData  dataWithContentsOfURL:url]; ❼
    //处理数据,在编辑器中渲染,让用户编辑
}

-(void)documentPickerWasCancelled:(UIDocumentPickerViewController *)controller { ❽
    //或许可以展示一条信息,表明用户没有选择文件
}

@end
```

❶ 编辑器应用的视图控制器遵守 UIDocumentPickerDelegate 协议。

❶ 编辑器可以处理的 UTI 类型。

❸ UIDocumentPickerViewController 对象使用 UTI 类型进行配置且处于 UIDocumentPickerModeImport 模式。

❹ 指定委托（这是必需的步骤）。

❺ 呈现视图控制器。
❻ 当用户选择文档时，调用委托回调方法。
❼ `url` 是本地文件 URL。文档的内容将复制到应用的 tmp/DocumentPickerIncoming 文件夹中。
❽ 当用户取消选择文档时，调用委托回调方法。

使用文档选择器时的用户导航如图 8-17 所示。



图 8-17：文档导入的用户导航：(1) 默认情况下打开 iCloud，选择位置以便选择提供者；(2) 选择提供者；(3) 选择文档；(4) 在编辑器应用中处理文档

2. 提供文档

要想成为提供文档的数据源，你需要以下步骤。

(1) 创建 UI 以帮助用户选择文档。
(2) 将文档的内容传递给编辑器应用。

要想创建 UI，视图控制器必须是 `UIDocumentPickerExtensionViewController` 的子类。当用

户选择此应用作为位置时，该视图控制器提供的 UI 在编辑器应用中是可用的（见图 8-17）。

当用户选择要使用的文档时，如果需要，视图控制器必须从服务器下载内容，并将该文档以文件 URL 的形式提供给编辑器应用。例 8-8 展示了选择器扩展的典型代码。

例 8-8　选择器扩展——视图控制器

```
@interface HPDocumentPickerViewController
    : UIDocumentPickerExtensionViewController ❶

@end

@interface HPEntry ❷
@property (nonatomic, copy) NSString *filename;
@property (nonatomic, copy) NSString *serverPath;
@property (nonatomic, assign) NSUInteger *size;
@property (nonatomic, copy) NSString *uti;
@property (nonatomic, copy) NSURL *iconURL;
@end

@implementation HPDocumentPickerViewController

-(void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    //从服务器检索文件的元数据并更新
    //UITableView是很好的选择 ❸
}

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath { ❹

    HPEntry *selected = [self.allFiles objectAtIndex:indexPath.row];
    //如果需要,从服务器下载内容 ❺

    NSURL* localFileURL = [self.documentStorageURL
        URLByAppendingPathComponent:selected.filename]; ❻
    [self dismissGrantingAccessToURL:localFileURL]; ❼
}

@end
```

❶ 视图控制器类，`UIDocumentPickerExtensionViewController` 的子类；向用户提供可用文档和目标的列表。

❷ 表示远程文件条目的模型类。

❸ 从服务器检索文件列表、更新 UI、等待用户响应、显示加载指示器或进度条——所有与 UI 有关的花哨东西。

❹ 假设使用了 `UITableView`，当用户选择文件时，触发委托回调。

❺ 如果文件内容在服务器上，则必须由应用扩展下载。最终的内容必须来自本地文件 URL。

❻ 将内容保存到 self.documentStorageURL 文件夹中，该 URL 是在使用文件提供者时设置的。

❼ 一切都准备好了。通知操作系统，编辑器应用必须被授予使用该文件的权限。操作系统将文件复制到编辑器应用对应的文件夹中。编辑器应用中的文档选择器（传递者）会被通知到。我们已经在例 8-7 中讨论了整个故事的另一半。

8.4.5 应用群组

应用扩展是一项非常有趣的功能。虽然扩展总是与应用捆绑在一起，但它在自己的进程中运行，并有自己的数据沙箱。

在一个典型的场景中，主应用将与应用扩展连接，后者还可以与容器应用连接，如图 8-18 所示。

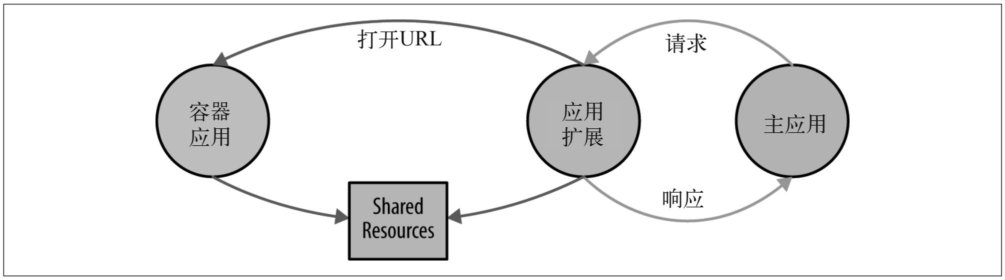


图 8-18：主应用、应用扩展和容器应用之间的通信（原始图像由苹果公司提供）

到目前为止，我们讨论的所有选项都是用于在多个应用间共享数据。然而，因为应用扩展在自己的沙箱中运行，所以它因此无法直接访问由容器应用直接存储的数据（用户默认值、文档文件夹、缓存文件夹、Core Data、SQLite，等等）。沙箱结构与图 8-19 所示内容类似。

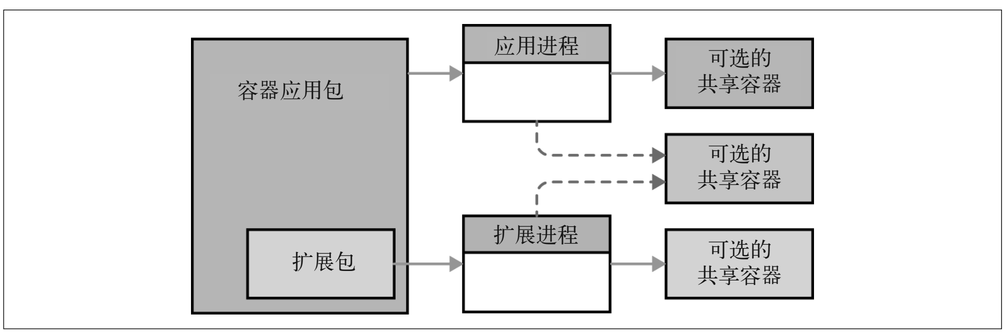


图 8-19：应用与应用扩展容器（原始图像由苹果公司提供）

iOS 7 引入的应用群组功能允许创建一个共享沙箱，容器应用和应用扩展都可以访问它。此外，应用群组支持在多个应用之间共享数据——但与共享钥匙串类似，应用必须使用相同的证书进行签名。

> 配置应用群组不仅是为了容器应用，还为了将所有需要共享容器的应用扩展绑定起来。可以配置多个应用群组，从而达到较为合理的数据隔离。

要想设置应用群组，转到项目清单，选择目标，然后在“功能”下将“应用群组”选项设置为“开”（见图 8-20）。最后，添加你将要使用的一个或多个群组 ID。

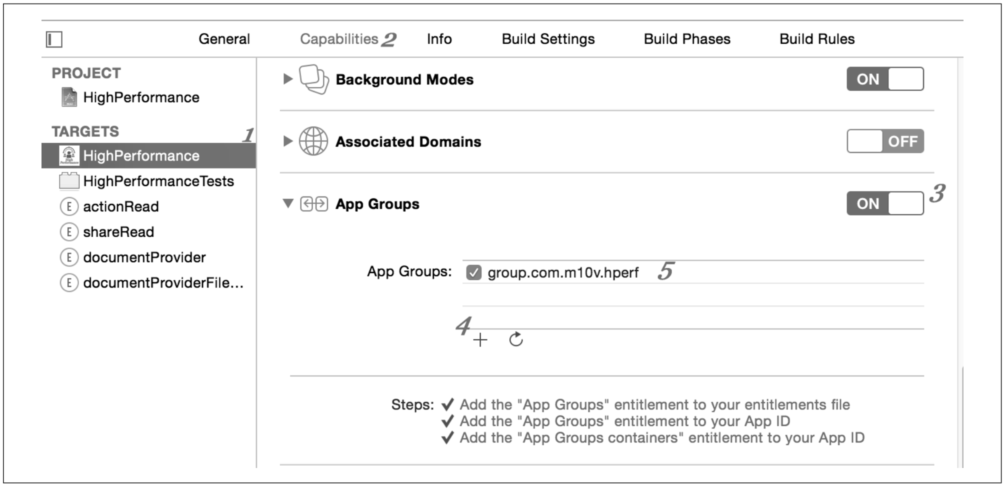


图 8-20：启用应用群组功能的 Xcode 设置

现在你可以使用 NSUserDefaults 或 NSFileManager 来处理共享数据，如示例 8-9 所示。

例 8-9　使用应用群组共享数据

```
-(void)sharedDataUsingAppGroups {
    NSString *sharedGroupId = @"group.com.m10v.hperf"; ❶

    NSUserDefaults *defs = [[NSUserDefaults alloc]
            initWithSuiteName:sharedGroupId]; ❷

    NSFileManager *fileMgr = [NSFileManager defaultManager];
    NSURL *groupFolder = [fileMgr
        containerURLForSecurityApplicationGroupIdentifier:sharedGroupId]; ❸
}
```

❶ 群组 ID；必须与清单中提供的匹配（见图 8-20）。

❷ NSUserDefaults；使用 initWithSuiteName: 初始化器。

❸ NSFileManager；使用 containerURLForSecurityApplicationGroupIdentifier: 方法获取共享文件夹。

> 如果需要访问网络，那么就使用 NSURLSession，以便容器应用和应用扩展可以访问传输的数据并共享网络的参数（特别是 cookie jar）。

8.5 小结

本章探讨了跨应用共享数据的几个选项。在所有可用选项中，自定义 URL scheme 是唯一允许从 Web（通过 Safari 浏览器或嵌入式 `UIWebView`）向原生应用共享数据的方案。但其他方案能够提供更加丰富的 UI。活动可以弹出共享表，让用户选择自己喜欢的应用来处理数据，这在社交共享中十分有用。

对于那些属于同一公司且使用同一证书进行签名的应用，你可以使用共享钥匙串共享数据，它可以发挥强大的作用。

本章末尾研究了 iOS 8 中引入的应用扩展，并介绍了它们如何在应用间帮助应用扩展文档交互和活动。应用组可以轻松地访问应用和扩展中的共享数据。

第 9 章

安全

应用可能会在未知的执行环境中运行，并通过未知的传输网络交换数据，因此，应始终将安全性作为首要任务之一，以便保护用户及应用的敏感数据。

越狱和常规设备都存在风险。例如，来自 JosiahsTech 的 YouTube 视频（https://youtu.be/1NTpi4NjkCE）就演示了修改广受欢迎的 Temple Run 游戏是何等地简单。

> **安全：足够是不够的**
>
> 安全是一个广阔的领域，仅凭一本书中的几页内容肯定不足以深入详解该话题。若想更深层次地研究，可以参阅更专业的内容：
>
> - Jonathan Zdziarski 的 *Hacking and Securing iOS Applications: Stealing Data, Hijacking Software, and How to Prevent It*（O'Reilly，http://amzn.to/1bnf44K）
> - Charlie Miller 的《黑客攻防技术宝典：iOS 实战篇》[1]
> - David Thiel 的 *iOS Application Security: The Definitive Guide for Hackers and Developers*（No Starch Press，http://amzn.to/1Hq15sc）

不论是通过代码的执行（例如，从 1024 位 DSA 密钥的加密密钥转为 2048 位 RSA 的加密密钥）还是通过用户干预（例如，引入双因素认证或应用 PIN），任何附加的安全层都会导致应用变慢。因此，在保证用户完成意图的前提下，你需要对添加的安全措施（会导致延迟）进行权衡。

本章将探讨应用安全方面的一些重点内容，而不会做深入的理论研究。我们将从以下角度探讨与安全性相关的方法。

注 1：此书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1068。——编者注

- 应用访问
  如何访问应用的安全、如何管理身份，以及其他相关的主题。
- 网络安全
  这包括与服务器通信相关的所有内容。
- 本地存储
  与设备上的所有数据相关。
- 数据共享
  让数据从其他应用进入你的应用或从你的应用流向其他应用。

9.1 应用访问

你的应用可以执行验证，也可以不执行验证。对于大多数游戏、新闻、实用和其他类似的应用而言，可能不需要身份验证。

本节将讨论一些选择，以便识别设备、用户、同一设备上的多个用户，以及同一或多个设备上的多个应用中的同一用户。

9.1.1 匿名访问

应用可能需要验证，也可能不需要验证。例如，不需要订阅的新闻应用可能永远都不需要认证。然而，为了提供个性化新闻或广告，就需要为设备创建唯一的标识符，比如 Yahoo! Digest News 应用。

有两个选项可用于识别设备：供应商的标识符（Identifier for Vendor，IDFV）和广告商的标识符（Identifier for Advertiser，IDFA）。下面我们将详细介绍这两者。

IDFV（http://apple.co/1xxe8oK）是设备上每个应用的持久唯一的标识符，用于向应用的供应商标识设备。应用包 ID 的一部分用于生成 IDFV，因此，即使应用来自同一家公司，IDFV 也可能不同。

使用 `-[UIDevice identifierForVendor]` 方法获取 IDFV。如果用户在设备重启后没有解锁，但应用已在后台任务执行期间或在推送通知时被唤醒了，那么该值可能为 `nil`。如果是这样，那么就稍后重试。例 9-1 显示了检索 IDFV 的一些简单代码。不要在主线程上执行此操作。

例 9-1　检索 IDFV

```
-(NSString *)idfv {
    UIDevice *device = [UIDevice currentDevice];
    NSUUID *rv = device.identifierForVendor;
    while(!rv) {
        [Thread sleepForTimeInterval:0.005];
        rv = device.identifierForVendor;
    }
    return rv.UUIDString;
}
```

在 iOS 6 中，IDFV 是从包 ID 的前两个部分创建的。因此，对包 ID `com.bundle.id.app1` 而

言，只有 com.bundle 会被使用。

iOS 7 中存在一个 bug，包 ID 分别为 com.bundle.id.app1 和 com.bundle.id.app2 的两个应用会有不同的 IDFV，即使它们来自于同一个供应商（使用相同证书来签署应用）。苹果公司并未修复此 bug，而是更新了文档。

在 iOS 7 和更新版本中，除了最后一部分，整个软件包 ID 都用于生成 IDFV。因此，IDFV 的生成表如表 9-1 所示。

表9-1：包ID的部分内容用于生成IDFV

包ID	iOS 6	iOS 7及更新版本
com.bundle.id.app1	com.bundle	com.bundle.id
com.bundle.id.app2	com.bundle	com.bundle.id
com.bundle.id.suite.app1	com.bundle	com.bundle.id.suite
com.bundle.id.suite.app2	com.bundle	com.bundle.id.suite
simpleid[2]	simpleid	simpleid

iOS 7 中的更改意味着，现在你可以使用两个选项来保留唯一的设备 ID，并跨多个应用进行跟踪。其中一个选项是，除最后一部分，保持包 ID 唯一。如果此种方式不可行，你可以选择第二个选项——使用共享的钥匙串来共享第一个已安装应用所获得的密钥。

当来自设备且属于同一供应商的所有应用都被卸载时，IDFV 会被重置。因此，如果只有一个应用，多次卸载和重新安装会生成不同的 ID。

如果无法使用共享钥匙串或相同的最后一部分包 ID，那么你将无法在多个应用之间唯一地识别此设备。

IDFA 是可重置的标识符，在设备上的所有应用中是唯一的。正因为在众多应用中是唯一的，所以它才是*真正唯一的* ID。但是，IDFA 可以被用户重置。此外，苹果公司对它的使用设置了限制，你必须保证在提交应用到 iTunes Connect 审核时使用它。此 ID 只应由广告投放系统使用。同时，它还带有一个表明用户是否希望使用此 ID 的标签。根据文档（http://apple.co/1OyHGYa）所述，如果标签未启用，那么 IDFA 只能用于广告的*频次控制、归属、转换事件、估计唯一用户数、广告欺诈检测和调试*。

这也就是说，你可以用 IDFA 估算应用的唯一用户数，但不能确定特定用户。例 9-2 显示了检索 IDFA 的示例代码。不要在主线程上调用 API，因为返回的值可能为 nil，这会导致重试。与 IDFV 一样，这种情况是有可能发生的，例如，如果设备已重启，但用户尚未解锁设备。

例 9-2　检索 IDFA

```
-(NSString *)idfa {
    ASIdentifierManager *mgr = [ASIdentifierManager sharedManager];
    if(mgr.isAdvertisingTrackingEnabled) {
        UUID *rv = mgr.advertisingIdentifier;
        while(!rv) {
            [Thread sleepForTimeInterval:0.005];
```

注 2：不推荐此类包 ID。

```
            rv = mgr.advertisingIdentifier;
        }
        return rv.UUIDString;
    }
    return nil;
}
```

仍旧使用 UDID？

唯一设备标识符（unique device identifier，UDID）现已弃用（从 iOS 6 开始）。如果还没这样做，你应该删除对它的所有引用。

9.1.2　认证访问

当需要识别用户时，你需要认证访问。这并不意味着认证必须在你的应用中完成。以下是一些可用的认证选项。

- 应用密码

  它也被称为应用 PIN，无论是否存在登录至应用的一组凭据，应用 PIN 都是你想要添加到应用的本地凭据。实际上，它就是只存储在设备本地的密码。例如，费用管理应用可能永远都不会在服务器上存储任何数据，但仍希望保护在设备上的访问。另一方面，医疗记录应用可能会将密码作为第二层的安全保障。因此，用户首先使用所需的凭据（通常是用户名 / 电子邮件和密码）登录，并将本地安全作为附加层添加。

  图 9-1 显示了两个应用，一个仅使用了本地凭据，另一个使用了应用 PIN 作为辅助安全措施。

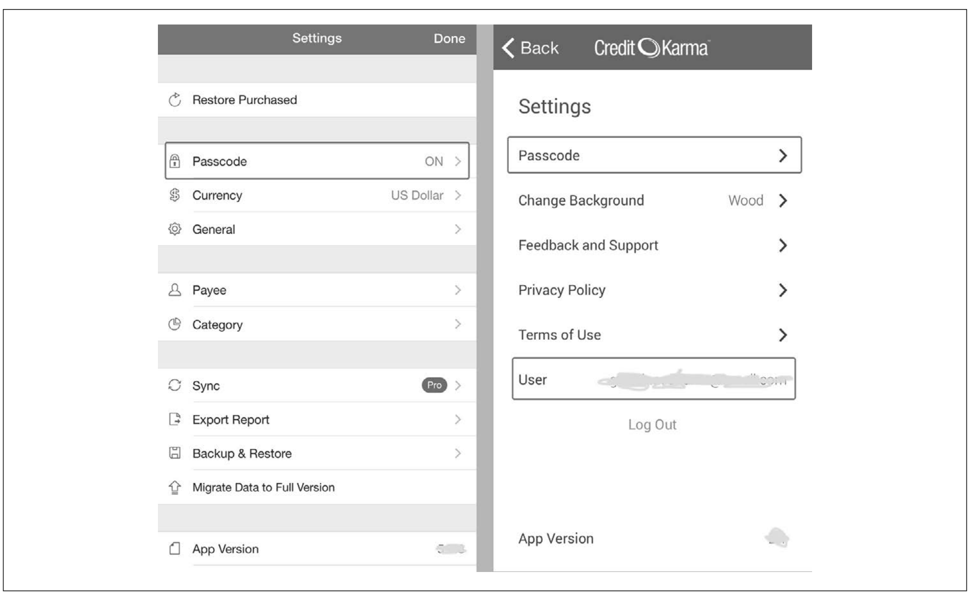


图 9-1：不使用远程凭证的 Expense report 应用（左）和使用应用 PIN 作为第二层安全性、有远程凭据的 Credit Karma 应用（右）

使用钥匙串在本地存储密码。不要在文件或数据库中以未加密的形式存储。

- 游戏中心
  此选项仅适用于游戏。用 GameKit 连接游戏中心，后者会负责使用凭据对用户进行验证。游戏中心可以访问用户资料、个人记录等，但仅共享唯一标识用户所需的内容（即用户 ID）。

  例 9-3 显示了在游戏中心登录后获取用户身份的模板代码。

例 9-3　使用游戏中心进行登录

```
#include <GameKit/GameKit.h> ❶

@implementation HPLoginViewController

-(void)authWithGameCenter {
    GKLocalPlayer __weak *player = [GKLocalPlayer localPlayer]; ❷
    if(!player.authenticated) { ❸
        player.authenticateHandler = ^(UIViewController *vc,
                                       NSError *error) { ❹
            if(error) {
                //处理错误
            } else if(vc) { ❺
                [self presentViewController:vc animated:YES completion:^{
                    //再次核实用户现在是否已经认证
                }];
            } else {
                GKLocalPlayer *lp = player;
                if(lp) {
                    [self verifyLocalPlayer:lp]; ❻
                }
            }
        };
    } else {
        [self verifyLocalPlayer:lp];
    }
}

-(void)verifyLocalPlayer:(GKLoalPlayer *)player {
    [player generateIdentityVerificationSignatureWithCompletionHandler:
        ^(NSURL *publicKeyURL, NSData *signature,
            NSData *salt, uint64_t timestamp,
            NSError *error) { ❼

        if(error) {
            //处理错误!
        } else {
            //player id = player.playerID
            //使用数据核实 ❽
```

```
            }

        }];
    }

    @end
```

❶ 引入 GameKit 头文件。不要忘记与 `GameKit.framework` 链接。

❷ `GKLocalPlayer` 表示运行游戏的已认证玩家。在任何时刻，设备上只能有一个认证玩家。

❸ 检查玩家是否通过认证。

❹ 设置 `authenticateHandler` 属性将触发授权。

❺ 如果用户以前没有授权与 GameKit 连接，那么在回调中返回的视图控制器必须显示出来。

❻ 如果没有视图控制器，也没有错误，则一切正常。但是需要做更多的工作以获取用户的详细信息。

❼ 使用 `generateIdentityVerificationSignatureWithCompletionHandler:` 方法获取签名，从而验证本地玩家。

❽ 实际的验证任务应该发生在服务器上（https://developer.apple.com/library/ios/documentation/GameKit/Reference/GKLocalPlayer_Ref/index.html#//apple_ref/occ/instm/GKLocalPlayer/generateIdentityVerificationSignatureWithCompletionHandler:），如下所述。

对于本地玩家的验证（服务器端），请遵循下列步骤。

(1) 使用 `publicKeyURL` 获取 X.509 证书。URL 必须是域名为 `apple.com` 的 `https` URL。此密钥必须由苹果公司签名。

(2) 按顺序连接 `player.playerID`、应用的包 ID、大字节 UInt 64 格式的 `timestamp` 和 `Salt`。

(3) 生成连接数据的 SHA-1 散列。

(4) 用步骤 1 中下载的公钥验证针对此散列的签名。

(5) 如果它们匹配，则一切正常。用户进行了验证，且 `player.playerID` 也可以使用。

- 第三方认证

  语义与游戏中心的身份验证类似，因为你拥有用户和登录体验。具体的 SDK 不在这里赘述，但你可以随时查阅 Facebook（https://developers.facebook.com/docs/ios）、Google+（http://bit.ly/gp-signin）或 Twitter（https://dev.twitter.com/twitter-kit/ios/twitter-login）的第三方认证 API。

- 你自己的验证

  大多数应用选择保留对注册和登录过程的完全控制，这需要自定义的认证机制。使用具有密码的电子邮件 / 用户名作为凭证是最常用的认证机制。我们简要讨论一下在实施这一过程中要采取的关键措施。

  ♦ 强制要求使用强密码，长度至少为六个字符，包含大小写字符（如果适用，可以使用罗马字母表）、数字和特殊字符。

    一些应用设置了最大长度的限制，但这一措施不是特别好。这好像在对用户说，“嗨！很抱歉，我们不允许有更高的安全性”。

此外，另一个重点是，较长但易于记忆的密码可能会比较短且模糊的密码更难破解。然而，因为在移动设备上输入较长的密码是比较繁琐的，所以通常选择较短和较复杂的密码。

- 提供活动会话的列表，并允许用户将其他设备或位置上的现有会话设为无效。
- 支持双因素身份验证，并在遇到异常行为时使用。例如，在异常时间从新位置或从新设备登录。
- 就与金融或金钱相关的应用而言，启用会话超时机制（例如，如果应用在后台停留了特定时间，如超过 5 或 10 分钟，则将会话设为无效）。这与启用网站上的“记住我”选项类似。
- 或者，在用户使用非到期访问令牌永久登录时，使用较短的应用 PIN 进行本地身份验证。图 9-1 展示了带有本地应用 PIN 的 Credit Karma 应用。
- 对于永久登录，确保访问令牌（类似于浏览器中的 Cookie）存储在本地的钥匙串中。
- 启用 CAPTCHA（你可以用限制性的方式使用此功能，例如，你可以在连续 3 或 5 次的无效尝试后启用）。
- 或者，使用本地身份验证将 Touch ID 与钥匙串集成，从而进行无密码登录。
- 请遵循 9.2.2 节和 9.3 节中讨论的最佳实践。

> 使用钥匙串或 Touch ID 这样的加密选项会增加开销，从而导致对用户的响应变慢。
>
> 即使使用最新的更新版本，Touch ID 也是缓慢且不可靠的，特别是可能出现指纹不被识别，导致多次重试。

9.2 网络安全

第 7 章对网络进行了深入的讨论。本节将讨论在与远程设备通信中与安全有关的最佳实践，该远程设备可以是服务器，也可以是点对点设备。

9.2.1 使用HTTPS

假设你将 HTTP 作为底层消息传递协议（TCP 是传输层协议），那么你必须通过 TLS/SSL 使用它。这也就是说，你应该一直使用 HTTPS。但是，使用 HTTPS 有几个问题。如果这些潜在风险未得到解决，则 HTTPS 可能会受到影响。

1. CRIME攻击

不要使用 SSL/TLS 压缩。如果你现在在使用，请在继续之前立即关闭它。这会让你处于较大的风险当中。使用 TLS 压缩（`gzip`、`deflate` 或其他格式），任何请求都会受到 CRIME（Compression Ratio Info-leak Made Easy，压缩率使信息很容易泄露）攻击。要想缓解风险，可以关闭 TLS 压缩，并给每个响应发送反 CRIME cookie，较为简单的方式是发送一个唯一的随机序列 cookie。

2. BREACH攻击

如果使用请求 / 响应正文压缩（`Transfer-Encoding = gzip` 或 `deflate`），你的通信会受到

BREACH（Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext，通过自适应超文本压缩的浏览器侦听和渗透）攻击，这种攻击类型于 2012 年 9 月首次发现。当满足以下标准时，就会存在风险。

- 应用使用 HTTP 压缩。
- 响应反映了用户输入。
- 响应反映了隐私。

没有单一的方法可以降低这种风险。The Breach Attack（http://www.breachattack.com）网站按有效性列出了以下方法。

- 禁用 HTTP 压缩。这种方法增加了传输的数据量，可能不会作为实际的解决方案。
- 从用户输入分离出隐私。将授权码放在远离请求正文的地方。
- 对每个请求进行随机化加密。但是，由于每个请求的加密是随机的，因此，多个并行请求可能无法实现了。
- 修饰隐私。不要以原始格式发送隐私。
- 使用 CSRF 保护易受攻击的 HTML 页面。在移动原生应用上，除非使用移动 Web，否则不需要 CSRF。
- 隐藏长度。一个较好的方法是在 HTTP 响应中使用分块传输编码。
- 对请求限速（这应该作为最后的方法）。

9.2.2　使用证书锁定

HTTPS 不是万灵药——采取 HTTPS 不会神奇地确保所有的通信都是安全的。HTTPS 的基础是对公钥的信任，该公钥用于加密初始消息（在 SSL 握手期间）。中间人（man-in-the-middle，MITM）攻击会捕获用于加密消息的密钥。

图 9-2 显示了中间人攻击的概要，其中中介器（如设备连接的 WiFi 热点或正在使用的代理服务器）拦截来自设备的请求。当设备发送对服务器证书的请求时，中介器将请求发送到服务器并捕获其应答。然后，它并没有将密钥返回至设备，而是返回了自己的密钥。这与 Charles 代理服务器使用的技术相同（参见 7.3.3 节）。

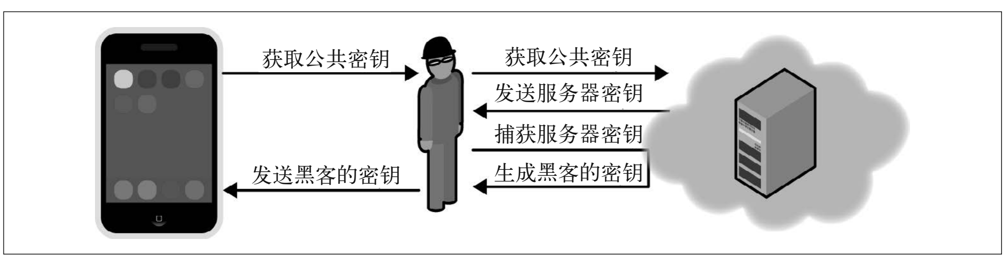


图 9-2：中间人攻击

不让请求变成无效的唯一方法就是*信任*，该信任由网络库放置在接收到的证书之中。证书只是签名的公钥。因此，如果网络库信任签名者，那它也会信任主机提供的公钥。黑客提供的假的*根证书*成为了让所有安全措施崩溃的罪魁祸首。

找到用于开发并安装了 Charles 证书的个人设备并不难。况且，因为私钥和证书在公共域中都是可用的，所以使用证书作为攻击的起点并不罕见。
市场上的二手设备有多少真的可以信任？越狱一台 iOS 设备是相当容易的。

这个问题的解决方案就是所谓的证书锁定（https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning）。这种方案的工作原理是，通过只信任一个或几个能够作为应用根证书的证书，应用创建一个自定义的信任级别。这允许应用仅信任来自白名单的证书，确保设备上永不安装那些允许网络监视的未知证书。

在使用 NSURLConnection 时，你可以提供一个执行证书验证的 NSURLConnection Delegate。

例 9-4 显示了可在应用中实现证书锁定的代表性代码[3]。

例 9-4　证书锁定

```
typedef void(^HPResponseHandler)(NSURLResponse *, NSError *error);

@interface HPPinnedRequestExecutor ❶

@property (nonatomic, readonly) NSURLRequest *request;
@property (nonatomic, copy) HPResponseHandler handler;

@end

@interface HPPinnedRequestExecutor () <NSURLConnectionDelegate>

@property (nonatomic, readwrite) NSURLRequest *request;

@end

@implementation HPPinnedRequestExecutor

-(instancetype)initWithRequest:(NSURLRequest *)request {
    if(self = [super init]) {
        self.request = request;
    }
    return self;
}

-(void)executeWithHandler:(HPResponseHandler)handler {
    self.handler = handler;
    [[NSURLConnection alloc] initWithRequest:self.request delegate:self];
}

- (void)connection:(NSURLConnection *)connection
    didReceiveResponse:(NSURLResponse *)response {
    //做常规的事情,用处理器发送结果
}
```

注 3：改编自 OWASP（https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning#iOS）。

```
-(BOOL)connection:(NSURLConnection *)connection
    canAuthenticateAgainstProtectionSpace:(NSURLProtectionSpace*)space {
    return [NSURLAuthenticationMethodServerTrust
        isEqualToString:space.authenticationMethod]; ❷
}

- (void)connection:(NSURLConnection *)connection
    didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *)challenge {❸

    void (^cancel)() = ^{
        [challenge.sender cancelAuthenticationChallenge:challenge];
    };

    if([NSURLAuthenticationMethodServerTrust
        isEqualToString:challenge.protectionSpace.authenticationMethod]) {

        SecTrustRef serverTrust = challenge.protectionSpace.serverTrust;
        if(serverTrust == nil) {
            cancel();
            return;
        }

        OSStatus status = SecTrustEvaluate(serverTrust, NULL);
        if(status != errSecSuccess) {
            cancel();
            return;
        }

        SecCertificateRef svrCert = SecTrustGetCertificateAtIndex(serverTrust, 0);
        if(svrCert == nil) {
            cancel();
            return;
        }

        CFDataRef svrCertData = SecCertificateCopyData(svrCert);
        if(svrCertData == nil) {
            cancel();
            return;
        }

        const UInt8* const data = CFDataGetBytePtr(svrCertData);
        const CFIndex size = CFDataGetLength(serverCertificateData);
        NSData* cert1 = [NSData dataWithBytes:data length:(NSUInteger)size];

        if(cert1 == nil) {
            cancel();
            return;
        }

        NSString *file = [[NSBundle mainBundle]
            pathForResource:@"pinned-key"
            ofType:@"der"];
        NSData* cert2 = [NSData dataWithContentsOfFile:file];

        if(cert2 == nil) {
```

```
            cancel();
            return;
        }

        if(![cert1 isEqualToData:cert2]) {
            cancel();
            return;
        } ❹

        [challenge.sender
            useCredential:[NSURLCredential credentialForTrust:serverTrust]
            forAuthenticationChallenge:challenge]; ❺
    }
}

@end
```

❶ 实现 NSURLConnectionDelegate 协议。

❷ connection:canAuthenticateAgainstProtectionSpace: 方法检查委托是否能够响应保护空间的身份验证形式。对于 SSL（服务器信任），返回 YES。

❸ connection:didReceiveAuthenticationChallenge: 方法处理 challenge，可以取消认证（无效时）或使用凭证（有效时）。

❹ 如果发生了失败，则将证书置为无效，与找不到证书或与绑定的密钥不匹配时一样。

❺ 如果一切都成功了，将证书置为可用。

iOS 8 及更新版本的注释

委托回调方法 connection:canAuthenticateAgainstProtectionSpace: 和 connection:didReceiveAuthenticationChallenge: 在 iOS 8 中被弃用，改为 connection:willSendRequestForAuthenticationChallenge: 回调。

根据验证结果，委托需要调用 NSURLAuthenticationChallengeSender 协议的以下方法之一（使用 challenge.sender 对象）。

- useCredential:forAuthenticationChallenge:
  验证成功。使用证书。
- cancelAuthenticationChallenge:
  验证失败。取消请求。
- continueWithoutCredentialForAuthenticationChallenge:
  没有证书也继续（不要这样做）。
- performDefaultHandlingForAuthenticationChallenge:
  让请求通过系统提供的默认路由进行。
- rejectProtectionSpaceAndContinueWithChallenge:
  拒绝当前提供的保护空间。这种情况比较少见，如果有的话，则用于 SSL 证书验证。

类似的方法可用于实现 `NSURLSession` 对象。[4] 这允许在会话的范围内进行控制，而不必担心应用创建的每个请求。

> 你不必维护大量的代码。像 `RNPinned CertValidator`（https://github.com/rnapier/RNPinnedCertValidator）这样的库有助于将代码减少到几行（http://robnapier.net/pinning-your-ssl-certs）。
> 如果使用 `AFNetworking` 库，你可能需要编写不同的代码。互联网上的一些教程 [5] 可以为你提供一些帮助。

9.3 本地存储

与通过网络交换的数据类似，存储在设备上的数据是不能防止被篡改的，而且如果不小心处理的话，入侵者是可以读取或修改数据的。以下是需要注意的几个要点，以及为了保护本地存储空间需要遵循的最佳实践。

- 本地存储不安全

  在越狱设备上非常容易访问本地存储。如果观看了本章开头引用的视频，你可能会注意到，使用一些即插即用的工具就可以替换或修改这些文件。

  你可能会说，“这只是众多设备中的一个，并且篡改数据也是为了设备上的用户”。我同意这种说法。但了解此类型的篡改对整个应用生态系统产生的副作用却是非常重要的。

  例如，在邮件应用中，设备可能会被注入用于发送邮件的一些数据，这会导致应用很容易就发送大量邮件。即使用户之后会被列入黑名单或被阻止，但伤害已经造成了。服务器应实现额外的安全性，使用速率限制技术和增强的 DDoS 保护措施来进行防卫。

- 加密本地存储

  本地存储可以利用操作系统提供的数据保护能力进行加密。

  要想启用数据保护，打开 Xcode，然后选择目标。在“功能”选项下，查找“数据保护”，并将其打开（见图 9-3）。这将为应用 ID 添加数据保护权利。

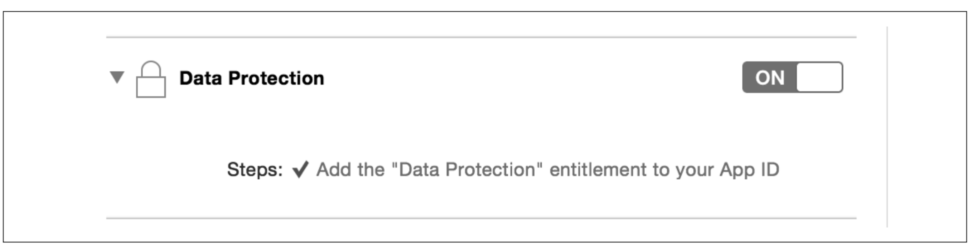


图 9-3：在 Xcode 中启用数据保护

注 4：iOS Developer Library,“Authentication Challenges and TLS Chain Validation”(http://apple.co/1ElV6Ud).

注 5：例如，Eric Allam 的“AFNetworking SSL Pinning with Self-Signed Certificates”（http://initwithfunk.com/blog/2014/03/12/afnetworking-ssl-pinning-with-self-signed-certificates/）。

在默认情况下，启用数据保护后，应用使用的所有本地存储都将使用设备密码进行加密。这意味着在设备解锁前无法访问数据。

你可以在 Apple Developer 门户上配置安全级别，导航至 Certificates，Identifiers & Profiles → Identifiers。转到应用 ID 子部分，然后选择要配置的应用 ID。你应该能注意到数据保护是启动状态（见图 9-4）。

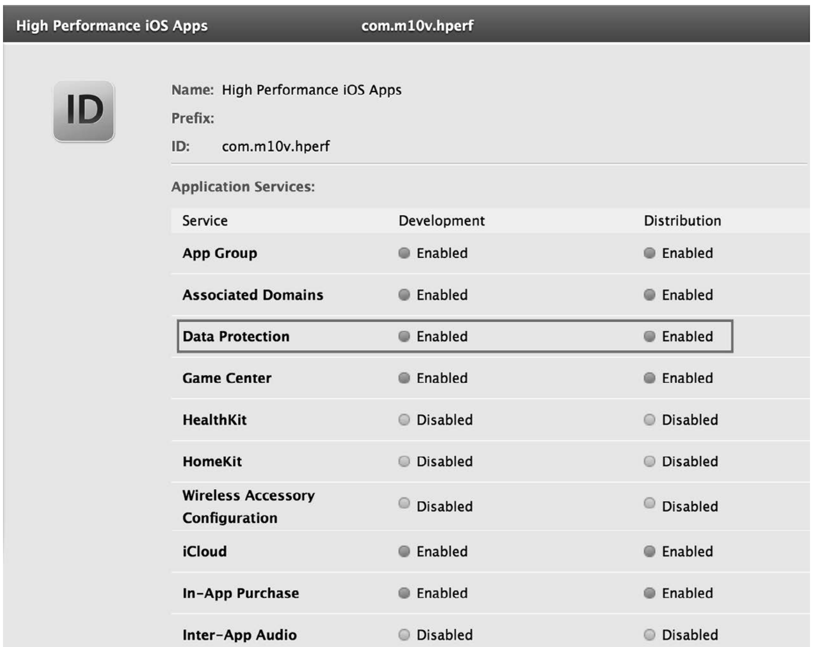


图 9-4：Apple Developer → App Capabilities Con guration → Data Protection

单击编辑按钮即可配置功能。图 9-5 展示了数据保护功能的安全级别。

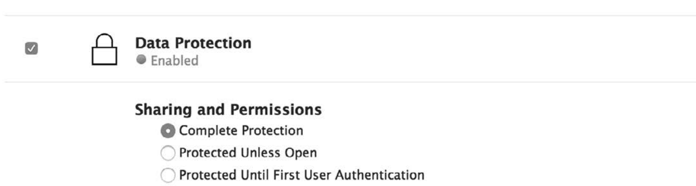


图 9-5：数据保护的安全级别

共享和权限选项如下。

- 完全保护：在任何时候访问文件进行读取或写入时，设备都需要是解锁状态的。设备锁定后不久（如果“需要密码”一项设置为“立即”时，则在 10 秒后），加密密钥会被废弃，导致所有数据无法访问，直到设备再次解锁。

- 打开前保护：这需要在创建文件句柄时解锁设备，一旦句柄可用，则设备锁定时也可以写入内容。这一选项在设备已经锁定，但还需要对文件进行写入时比较有用。例如，当设备已经被解锁，用户可以触发邮件应用来下载附件，而在设备被锁定之后，应用可以触发后台操作继续下载。
- 第一个用户认证前保护：这需要设备在重新启动后进行一次解锁。第一次解锁后，应用可以访问所有文件，并且没有任何限制。

  需要注意的是，无论安全级别是多少，一旦启用数据保护，重新启动后不能立即访问文件——用户必须至少将设备解锁一次。这也意味着，应用在接收推送通知后尝试读 / 写文件时，如果设备在重新启动后从未解锁，那么将导致错误。

  对于每个文件的保护，你可以使用方法 `- [NSData writeToFile:options:error:]`；对于完全保护，将选项设置为 `NSDataWritingFileProtectionComplete` ；对于打开前保护，设置为 `NSDataWritingFileProtectionCompleteUnlessOpen`；对于第一个用户认证前保护，设置为 `NSDataWritingFileProtectionCompleteUntilFirstUserAuthentication`。注意，如果在应用级别启用了数据保护，则默认值与开发者门户上设置的一致。

  或者，你可以使用方法 `- [NSFileManager createFileAtPath:contents:attributes:]`，将属性字典设置为 `@{NSFileProtectionKey:<required-level>}`（如果需要，还有其他属性），其中 `required-level` 可以是 `NSFileProtectionComplete`、`NSFileProtectionCompleteUnlessOpen` 或 `NSFileProtectionCompleteUntilFirstUserAuthentication`。

  此外，使用属性 `- [UIApplication protectedDataAvailable]` 确定是否可以访问受保护的文件。如果设备未锁定或未启用数据保护，则该值会设置为 `YES`。当修改属性值为 `NO` 时，具有 `NSFileProtectionComplete` 或 `NSFileProtectionCompleteUnlessOpen` 属性的文件无法被访问，直到设备解锁，同时，具有 `NSDataWritingFileProtectionCompleteUntilFirstUserAuthentication` 属性的文件也无法被访问，直到设备重新启动，并解锁。

- `NSUserDefaults` 不安全
  通常来说，我们总是认为用户默认值被保存在安全的地方。事实上，它们很简单。plist 文件是与其他应用文件一起保存的。[6]
- `NSBundle` 值也不安全
  噢！应用包的设置被当作是与应用捆绑在一起的，从不修改。这种理解只有部分是正确的，因为包含值的 .plist 文件实际上是可以被篡改的。
- 不要完全依靠钥匙串
  钥匙串的安全可以被破坏。攻击者在设备锁定时肯定无法访问关键信息。然而，重要的是不要过分依赖钥匙串。因为加密密钥是通过使用仅为 4 位数字的设备密码，经过预定公式产生的。所以这仅提供了最多 10 000 种组合——大家都知道，当涉及安全性时，10 000 种组合根本不算是强大的安全防护。考虑到 iOS 设备在 6 次错误尝试后，用户在一分钟内不允许进行登录（https://support.apple.com/en-us/HT204306），因此，10 000 种组合可以在几个小时内手动完成。

注 6：Prateek Gianchandani, InfoSec Institute,“iOS Application Security Part 20 - Local Data Storage (NSUser Defaults, CoreData, Sqlite, Plist Files)”(http://resources.infosecinstitute.com/ios-application-security-part-20-local-data-storage-nsuserdefaults-coredata-sqlite-plist-files/).

但是，一般来说，攻击者不必花费那么多时间。根据几年前发布的一份报告（http://danielamitay.com/blog/2011/6/13/most-common-iphone-passcodes），前 10 个密码的使用量占据了当前正在使用的密码总数的 15%。

好在最近的 iOS 版本有一个可以启用更强密码的选项。你可以进入 Settings → Touch ID & Passcode，然后你会发现 Simple Passcode 选项是被选中的（见图 9-6）。

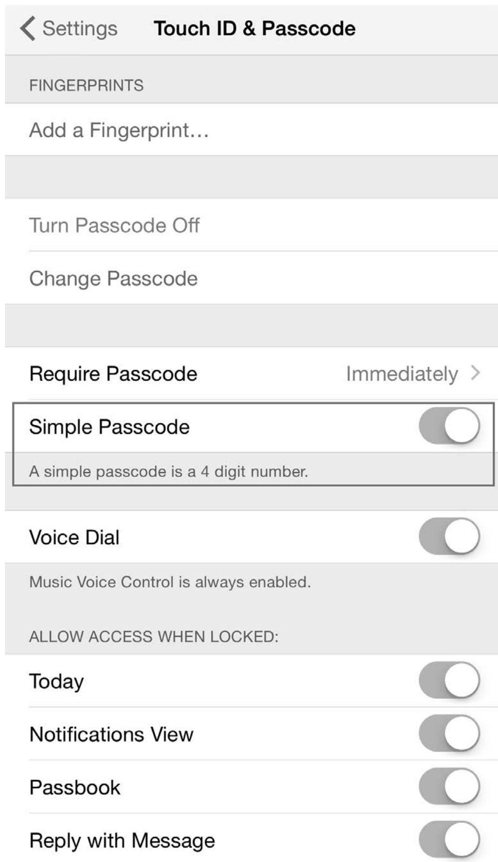


图 9-6：简单密码

简单密码是由 4 位数字组成的，而非简单密码可以是任意长度的，并且可以包括字母、数字、字符以及特殊字符。如图 9-7 所示，每个选项的键盘都不同。虽然使用字母数字密码是一个不错的选择，但启用此选项的用户总数非常少，所以不能依赖它。

作为一种可行的做法，你可以对存储在密钥库中的数据进行加密，并只存储最少的数据。还可以针对每个设备生成密钥，并在本地存储。再次强调，最好的做法是让攻击者更加难以定位和解密数据。如果攻击者可以物理访问设备，那你能做的也就只有这么多了。

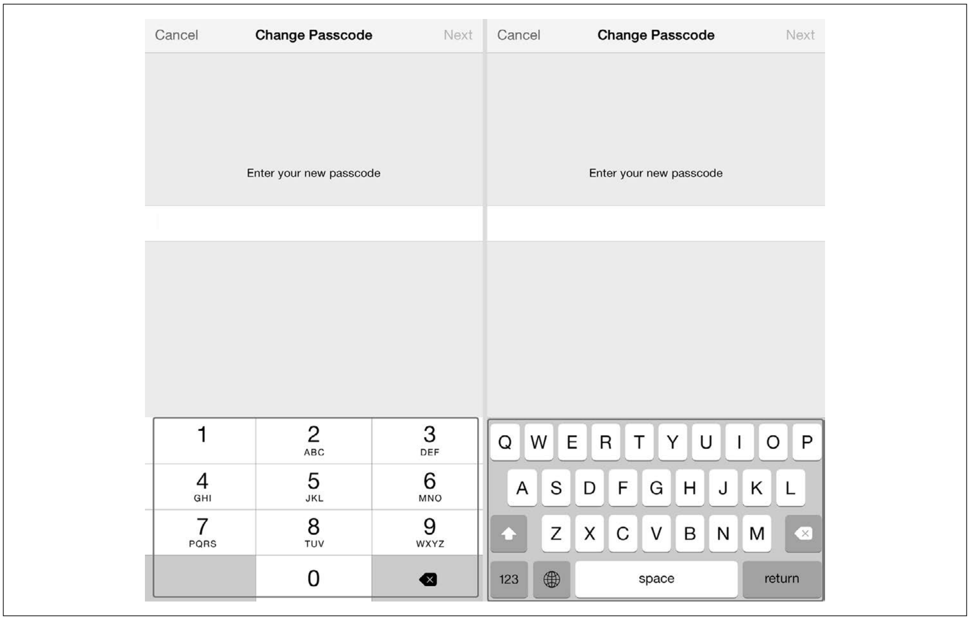


图 9-7：简单的密码使用 4 位数字（左），而复杂的密码可以使用字母数字和特殊字符（右）

- 小心记录的东西

  使用内置的 `NSLog` 函数进行日志记录是很常见的，因为这是开发人员被教导的方式。[7] 官方文档（https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Miscellaneous/Foundation_Functions/#//apple_ref/c/func/NSLog）声明了“将错误消息记录到苹果系统日志工具”的函数。它不是控制台日志记录。而且也没有 iOS 设备控制台。你甚至可以在记录的很多天后看到这些日志。

  在 Xcode 中，导航到 Window → Devices，你应该能看到连接设备和模拟器列表，如图 9-8 所示。



图 9-8：设备概览

注 7：从技术上讲，这不是数据存储。但设备日志类似于数据存储，具有自动和永久的特点。

如果单击 View Device Logs 按钮，你应该可以看到所有的日志，如图 9-9 所示。这包括 NSLog 的输出。

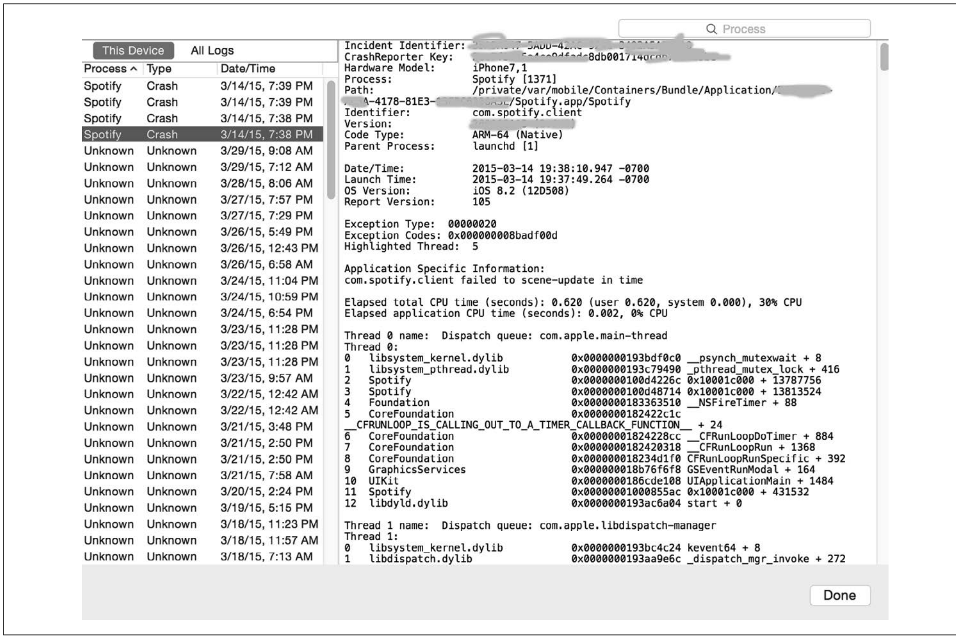


图 9-9：设备日志

作为最佳实践，不要在非调试版本中使用 NSLog。一个较好的方法是使用包装函数和宏，如例 9-5 所示。更好的选择是使用 CocoaLumberjack 这样的第三方库。我们已经在 1.4.4 节中深入讨论了日志记录。

例 9-5　使用 NSLog 记录日志

```
@implementation HPLogger

+(void)log:(NSString *)format, ... {

#ifdef _DEBUG ❶
    va_list args;
    va_start(args, format);
    NSLogv(format, args);
    va_end(args);
#endif

}
```

❶ 仅在调试模式下打出日志。你可以随意使用一些其他条件（而不是这里使用的 #ifdef _DEBUG），适合对应的应用即可。

安全的底线是，如果设备被解锁，则可以访问所有数据。即使设备被锁定，也可以访问大多数（也可能是所有）数据。

这是警察和小偷之间的比赛。人们只能期望两边的工具能更高级和更智能，但争权夺霸的比赛将永远不会结束。这一切归根结底就是谁来领导和谁来追赶。

9.4 数据共享

共享数据和处理传入数据时遵循的简单基本规则是：不要信任对方。

当接收数据时，总是进行验证。应用对数据的唯一假设应该是，它可能是无效且错误的。为了提高安全性，要求数据进行签名。

同样，因为不知道哪个应用会处理数据，所以永远不要发送敏感数据。如果你确实需要共享敏感数据，那么提供令牌，然后要求其他应用从你的应用（或服务器）请求数据。

9.5 安全和应用性能

额外添加的加密或安全措施会计入总内存的消耗之中，同时还会增加处理时间。你没有办法在所有维度上进行优化，只能做一些权衡。

有时，并非必须使用 2048 位的 RSA 密钥，1024 位的 DSA 密钥也许就已经足够了。其他时候，Rijndael 这样的对称加密算法就足以保护数据的安全了。[8]

从钥匙串检索初始值可能会导致加载时间延长。你在使用时应该小心谨慎。

证书锁定有其自己的成本，有可能会减慢所有的网络操作。

创建和验证数据签名需要计算内容哈希，这意味着会产生额外的内容传递。根据内容的大小，这可能需要较多时间，更不用说计算和验证数字签名所需的额外时间了。

所有这些步骤会快速叠加起来。也许你有了世界上最保险和最安全的应用，但如果仅加载程序就需要 30 分钟，估计也没有人想使用它。对于这一点，即使 5 秒钟都可能对用户体验产生负面影响，甚者永远失去用户，尤其在其他应用可以满足同样需求的情况下。

9.6 清单

为了保卫应用免受攻击，你需要了解有关恶意攻击和其他相关领域的知识，这些知识是无法完全列举出来的。在测试 iOS 应用的安全性时，你应该对照表 9-2 中的清单。

表9-2：安全清单

描述	状态
静态代码分析	
是否使用了 NSLog	YES/NO
如果是，仅在调试模式中使用 NSLog	YES/NO
所有的 URL 都是 HTTPS	YES/NO

注 8：注意，使用加密技术会使应用在 App Store 中的审批生效过程变得更长，因为可能会对你使用的 API 产生导出限制。

（续）

描述	状态
本地文件的路径不是硬编码的	YES/NO
检查最新版本和补丁程序的依赖关系	YES/NO
没有使用私有 API	YES/NO
代码中没有嵌入私钥或隐私	YES/NO
资源中没有嵌入私钥或隐私	YES/NO
没有运行不到的代码或无用代码	YES/NO
权利是正确的（没有丢失，没有附加）	YES/NO
如果使用 `connection:willSendRequestForAuthenticationChallenge:` 方法，没有 `useCredential:forAuthenticationChallenge:` 的直接分支（没有任何代码）	YES/NO
应用使用了 IDFV	YES/NO
应用使用了 IDFA	YES/NO
为应用签名设置正确的配置文件 / 证书	YES/NO
检查 SQL 注入	YES/NO
运行时分析——日志	
仅对文件执行日志记录	YES/NO
定期删除日志文件	YES/NO
执行日志循环	YES/NO
日志中没有隐私或敏感信息	YES/NO
打印栈跟踪时不会记录敏感信息 [9]	YES/NO
运行时分析——网络	
只使用 HTTPS URL	YES/NO
服务器有针对 CRIME 攻击的实现	YES/NO
服务器和客户端应用有针对 BREACH 攻击的实现	YES/NO
客户端使用证书锁定	YES/NO
设置正确的缓存策略	YES/NO
运行时分析——认证	
应用使用第三方身份验证	YES/NO
应用使用自定义身份验证	YES/NO
第三方验证 SDK 按照此清单的其余部分进行了严格的审核	YES/NO
登录 UI 隐藏了密码	YES/NO
密码不可复制	YES/NO
应用实现密码	YES/NO
密码存储在钥匙串中	YES/NO
可以通过服务器的配置更改身份验证工作流 [10]	YES/NO
运行时分析——本地存储	
应用使用本地存储	YES/NO
加密所有的敏感信息	YES/NO
周期性地清理存储	YES/NO

（续）

描述	状态
运行时分析——数据共享	
应用使用共享密钥库保存常见设置	YES/NO
验证深层链接 URL	YES/NO
验证任何传入的数据	YES/NO
不向未知应用共享任何敏感数据	YES/NO
使用应用扩展时，设置正确的群组 ID	YES/NO

此清单是以过去的安全性知识和以下资源为来源而编写的：

(1) 苹果公司的 iOS 安全文档（http://apple.co/1I6xVi1）
(2) OWASP 移动安全项目——安全测试指南（https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Security_Testing_Guide）
(3) OWASP iOS 应用安全测试备忘录（https://www.owasp.org/index.php/IOS_Application_Security_Testing_Cheat_Sheet）
(4) OWASP iOS 开发者备忘录（https://www.owasp.org/index.php/IOS_Developer_Cheat_Sheet）
(5) Stack Overflow，"iOS 6 安全分析工具"（http://security.stackexchange.com/questions/23564/security-analysis-tools-for-ios-6）
(6) iPhone 应用的渗透测试：第 1 部分（http://www.securitylearn.net/2012/02/12/penetration-testing-of-iphone-applications-part-1/）和第 2 部分（http://www.securitylearn.net/2012/04/20/penetration-testing-of-iphone-applications-part-2/）

9.7 小结

仅仅阅读单个章节是无法完全理解安全性的。本章从多个角度出发，简要介绍了安全方面的一些关键概括点。我们探究了在实施安全措施时的一些做法，并阐述了它们是如何影响整体的应用体验的。

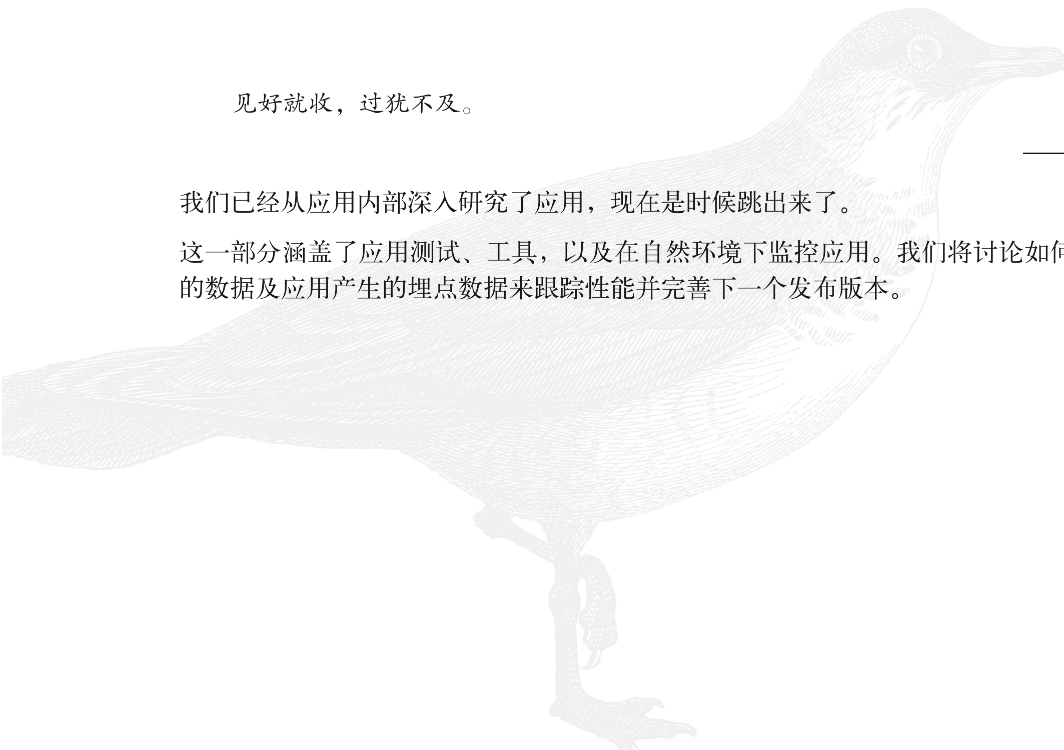
你在开发时应遵循本章末尾的清单，确保在应用中、在服务器上以及在两者之间能够解决常见的安全漏洞，或者至少能够实施明确有效的措施。

注 9：处理异常或其他情况时。

注 10：如果登录过程被破坏，应该能够改变认证流程。例如，触发双因素身份验证，添加 CAPTCHA，或在极端情况下从原生的登录 UI 切换到 Web 登录。

第四部分

代码之外



见好就收，过犹不及。

——佚名

我们已经从应用内部深入研究了应用，现在是时候跳出来了。

这一部分涵盖了应用测试、工具，以及在自然环境下监控应用。我们将讨论如何利用获得的数据及应用产生的埋点数据来跟踪性能并完善下一个发布版本。

第10章

测试及发布

测试一项功能、一个组件或一个应用与实现它同样重要。

开发团队编写代码来涵盖各种场景，而质量保证团队确保代码能够如期运行。质量保证团队和开发团队往往有重合之处，例如，在创业团队或小公司里，开发和质量保证工作常常由相同的人或团队来完成。

在本章中，我们将研究测试用例的基本概念、类型、支持类型的框架、测试自动化，以及持续集成。

假设团队遵循某种开发方法，甚至是牛仔式编码，那么你确实需要编写测试用例来规范化应用测试。

10.1 测试类型

*测试类型*与其预期的明确目的有关。例如，如果目的是测试一个方法、类或组件，那么它可能是单元测试。类似地，如果其目的是测试应用从安装部署到所有的功能，那么它可以归类为验收测试或端到端测试。

与其罗列各种各样的测试类型，不如关注以下类型。它们在面向用户的应用中是至关重要的类别，尤其是 iOS 应用。

- 单元测试
  在模拟环境中测试一个独立方法来保证其有效性。
- 功能测试
  在真实环境中测试一个方法来确保准确性。

- 性能测试
  测试一个方法、模块或完整应用的性能。

10.2　定义

以下的定义会在我们讨论测试时派上用场。

- 测试用例
  需要进行测试的一个场景。它包含一个方法、功能或程序执行所需的条件；测试场景需要的一系列输入变量；以及一个期望行为，其中包括系统的输出或改变。

- 测试夹具
  表示进行一个或多个测试用例前所需的准备和清理阶段。其中包括对象创建、依赖项设置、数据库设置，等等。

- 测试套件
  包含一系列测试夹具的测试用例，可以嵌套其他的测试套件。它用于聚合应当一起执行的测试用例。

- 测试运行器
  执行测试并提供结果的系统。在本书中，Xcode 是一个图形化的测试运行器。命令行工具同样可以启动一个测试自动化常用的客户端。

- 测试报告
  测试成功或失败的内容摘要，如有必要，还会附上错误信息。

- 测试覆盖率
  衡量测试套件进行的测试数量，并可以发现应用未被测试的部分。如图 10-1 所示，在代码层面测试时，测试覆盖率报告将总结有多少行代码被测试所覆盖。一份详尽的报告还可能标明哪部分代码是未经测试的。

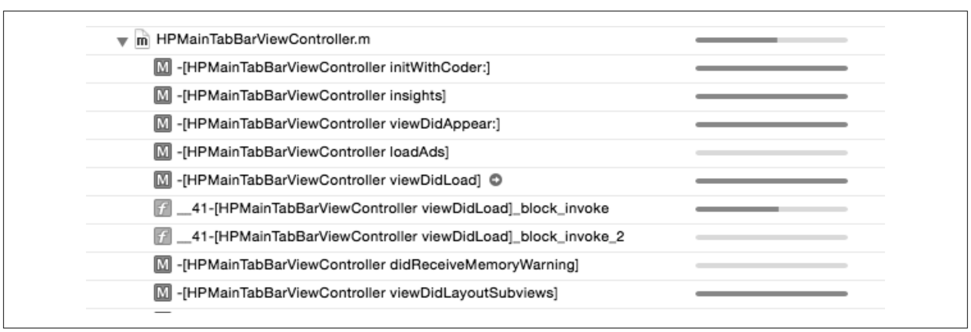


图 10-1：测试覆盖率报告

- 测试驱动开发
  测试驱动开发是一种反复迭代且开发周期很短的软件开发流程。其过程包含编写自动化测试用例、编写通过测试的最小代码集、重构代码以符合准入标准。

10.3 单元测试

单元测试检查单独的方法、一个或多个模块的集合，还包括隔离互相关联的数据以验证有效性。为实现隔离，依赖项将会以模拟数据的形式提供满足测试场景的行为。

虽然测试所有的方法（包括属性的获取方法和设置方法）相当枯燥，但这都是值得的。如果一个方法调用失败了，你将会得知是哪个方法出现了错误以及具体出错的原因。

Xcode 内置支持 XCTest（http://apple.co/1PlWsUa）单元测试框架。

10.3.1 设置

单元测试需要设定一个测试目标。如果工程中还没有设置一个测试目标，那么你需要依照以下步骤创建目标。

(1) 打开 XCode 中的测试导航器菜单。
(2) 点击 + 号按钮。
(3) 选择 New Test Target 选项。
(4) 如图 10-2 所示，输入新目标的内容。

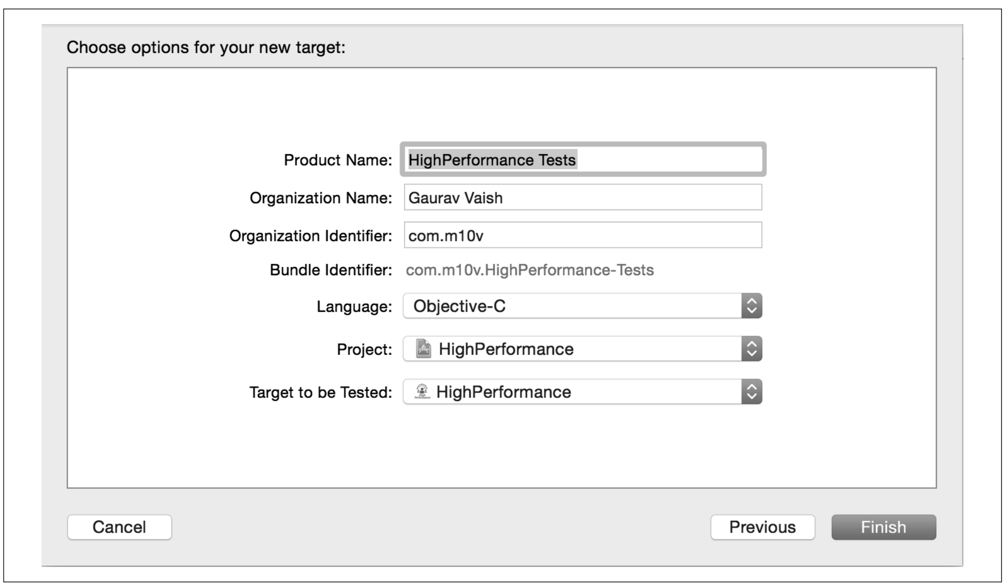


图 10-2：添加测试目标

Xcode 本应该在几处地方对工程进行配置。

首先，确认工程中已创建新目标（见图 10-3）。

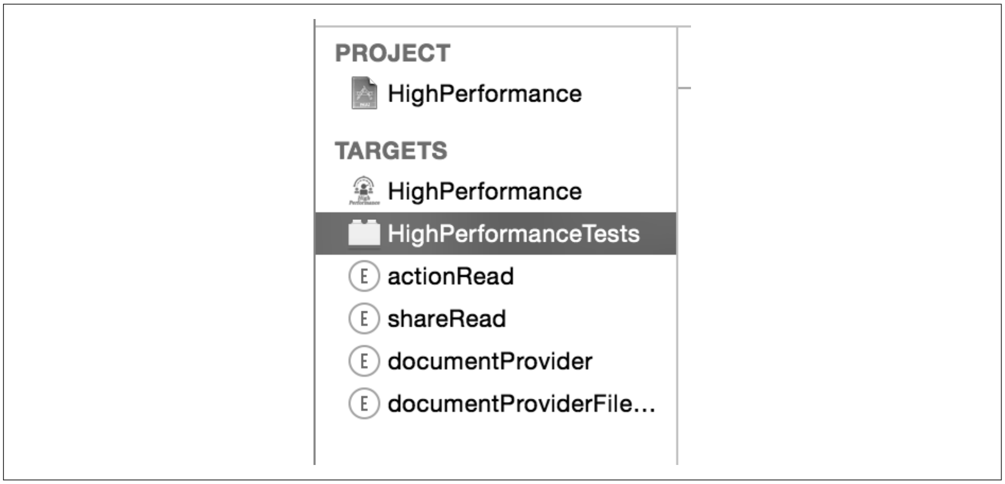


图 10-3：确认测试目标

其次，在 XCode 中通过 Product → Scheme → Manage Schemes → Select project → Edit 打开工程的 scheme，你应当能发现，Test → Test 处的一个条目与之前创建测试目标时输入的 Product Name 一致。如图 10-4 所示，在本例中是中 HighPerformanceTests。

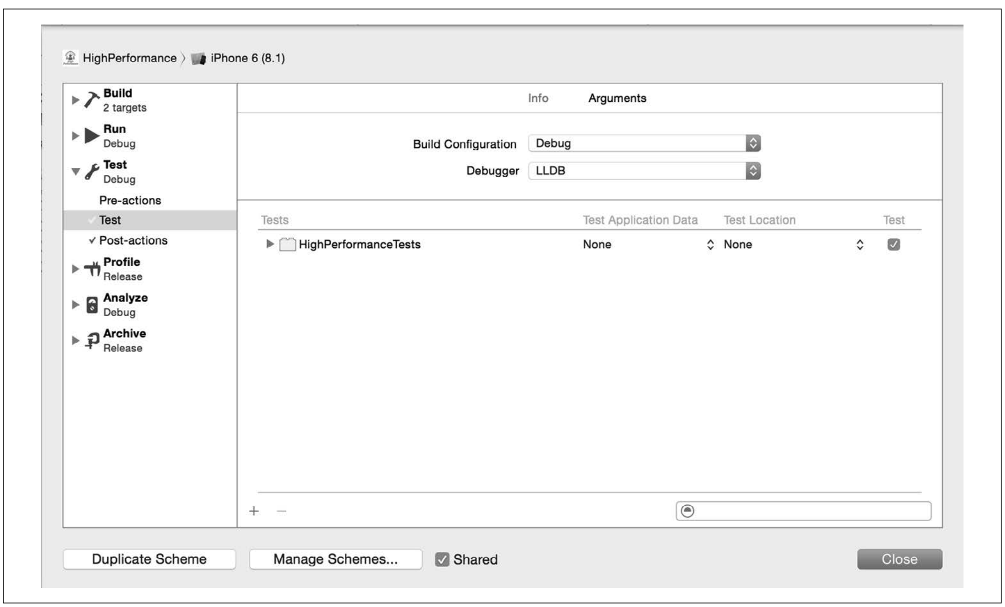


图 10-4：product scheme 测试设置

10.3.2 编写单元测试

创建目标后，我们可以开始编写第一个单元测试了。

XCTest 要求创建 XCTestCase 的一个子类来表示一个测试夹具（并非如其命名表示软件工作中的测试用例）。在该子类中编写测试用例。包含所有测试夹具的测试套件在运行时以特定顺序调用方法。

- +[setUp]
  这是测试夹具的设置方法，在类中所有的测试用例执行前被调用。测试夹具所有的通用初始化工作在此完成。注意，这是一个类方法。
- -[setUp]
  这是测试用例的设置方法。在每个测试用例运行前被调用。在此方法中完成每个测试用例的初始化工作。它是一个实例方法。
- -[testXXX]
  测试夹具的所有实例方法，名称以 test 开头，并且不含任何参数，都对应一个测试用例。
- -[tearDown]
  这是测试用例的清理方法，在每个测试用例执行完后被调用。它是一个实例方法。
- +[tearDown]
  这是测试夹具的清理方法，在类中所有测试用例执行完后被调用。它是一个类方法。

如果在执行测试方法时没有发生任何错误或异常，那么就表示测试用例是成功的。你可以通过 SDK 提供的断言宏检查更复杂的场景，比如测试 nil、测试等价性，等等。这些宏以 XCTAssert<AssertionType>[1] 形式命名。例如，XCTAssertEqual 用于测试两个值或对象是否等价。

图 10-5 展示了整个执行生命周期的可视化过程。注意，可以多次调用 -[setUp] 和 -[tearDown] 实例方法。

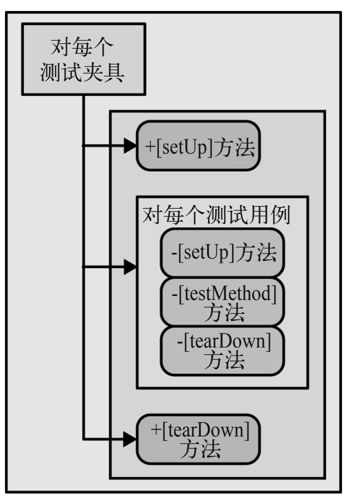


图 10-5：测试执行的生命周期

注 1：iOS Developer Library,“Assertions Listed by Category”(http://apple.co/1N2sS6g).

前文的例 4-9 中介绍了 HPAlbum 类，现在我们为此类写一个单元测试。

例 10-1　HPAlbum 单元测试

```
@implementation HPAlbumTest ❶
- (void)testInitializer { ❷
    HPAlbum *album = [[HPAlbum alloc] init];
        XCTAssert(album, @"Album alloc-init failed"); ❸
}

- (void)testPropertyGetters { ❹
        HPAlbum *album = [[HPAlbum alloc] init];
        album.name = @"Album-1";
        NSDate *ctime = [NSDate date];
        album.creationTime = ctime;

        HPPhoto *coverPhoto = [[HPPhoto alloc] init];
        coverPhoto.album = album;

        album.coverPhoto = coverPhoto;
        NSArray *photos = @[coverPhoto];
        album.photos = photos; ❺

        XCTAssertEqualObjects(@"Album-1", album.name);
        XCTAssertEqualObjects(ctime, album.creationTime);
        XCTAssertEqualObjects(coverPhoto, album.coverPhoto);
        XCTAssertEqualObjects(photos, album.photos); ❻
}

@end
```

❶ 测试夹具通常将类以类名＋ Test 的形式命名。

❷ 测试用例是一个实例方法，其名前缀是 `test`，方法名表示测试的内容。

❸ `XCTAssert` 方法用于断言对象不是 `nil`。

❹ 另一个测试用例，用于测试属性的获取方法。不要在一个测试用例中测试多个方法。

❺ 测试状态前的对象设置和需要提前执行的代码。

❻ 测试对象等价性的断言。如果断言失败了，则测试用例不通过。这说明代码中存有 bug，需要修复问题。

10.3.3　代码覆盖率

单元测试很重要，但如何在结束测试后得知代码中经过测试的部分和未经测试的部分？在自动化单元测试或功能测试中，我们使用*代码覆盖率*来表示经过测试的代码的百分比。

代码覆盖率文件可以直接经过苹果的 LLVM 代码生成器产生，并且你可以在 Xcode 中修改选项。有两种方式可以达到此效果，其一允许 Xcode 生成可视化报告，另一种生成基于 XML/HTML 的报告。

1. 集成覆盖率报告

开发人员可以运行测试用例并通过 Xcode 查看可视化报告。

要想获得 Xcode 的代码覆盖率报告，在主目标中开启测试覆盖率数据。如图 10-6 所示，通过菜单进入 Product → Scheme → Edit Scheme（⌘ <），在 scheme editor 对话框中，选择左侧的 Test 项后选中 Gather coverage data，之后点击 Done 按钮保存设置。

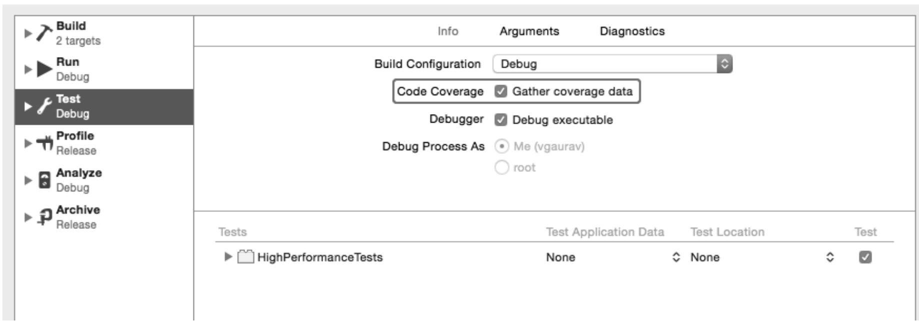


图 10-6：在 Xcode 中启用覆盖率数据收集

然后运行测试用例。根据设置，在运行测试用例的过程中，Xcode 会收集覆盖率的详细数据。你可以通过以下步骤查看覆盖率报告。

(1) 打开导航栏中的报告导航栏，如图 10-7 所示。

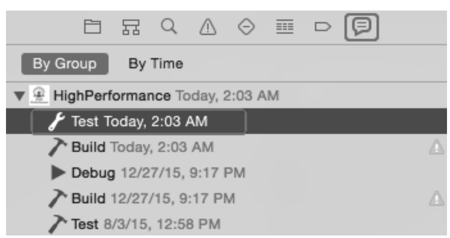


图 10-7：报告导航栏

(2) 选择最近一次运行的测试。
(3) 打开覆盖率标签。

你应该可以看到类似图 10-8 所示的覆盖率报告。

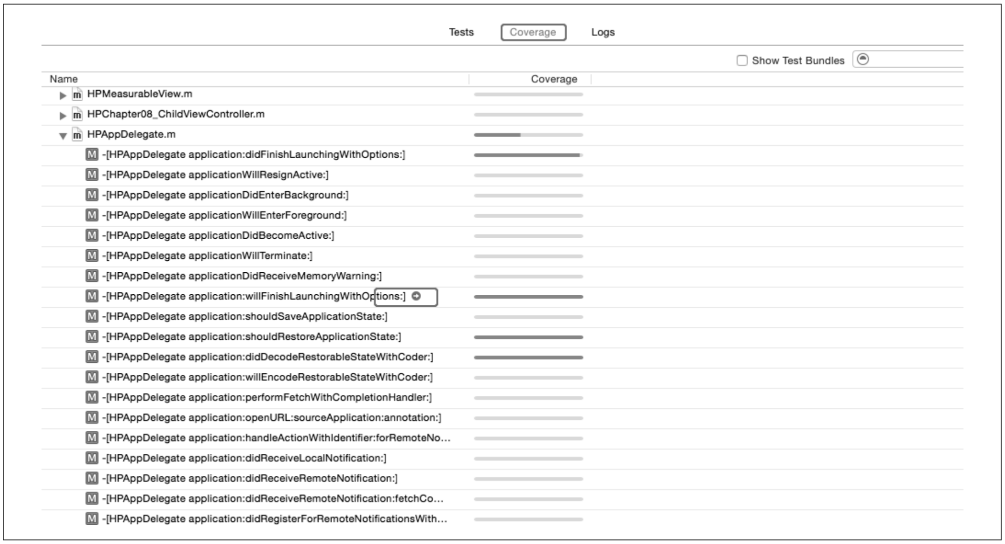


图 10-8：Xcode 集成的测试覆盖率报告

如果点击任意方法旁小的右箭头（见图 10-8），那么将会跳转到源代码处。跳转后你可以看到测试通过的确切代码行数以及未被测试覆盖的代码行数，如图 10-9 所示。

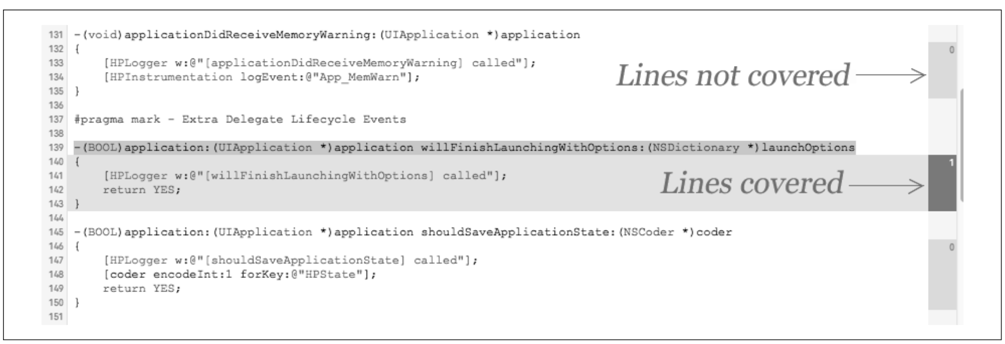


图 10-9：Xcode——源代码集成的覆盖率报告

2. 外置测试率报告

你也可以生成 XML 或 HTML 格式的报告。当你在持续集成环境或非开发人员的机器上运行测试时，或当你想要保存报告以备未来使用时，这非常有用。

为了生成包含覆盖率数据的报告，你需要启用以下标记。

- Generate Debug Symbols
  在编译生成的库中引入调试符号。

- Generate Test Coverage Files
  生成包含覆盖率数据的二进制文件。

- Instrument Program Flow
  在测试用例运行时检测应用。

推荐使用自定义的构建设置来区别测试覆盖率和常规构建，因为前者可以在常规测试中缓慢进行。图 10-10 展示了你可以在哪里找到这些构建设置。

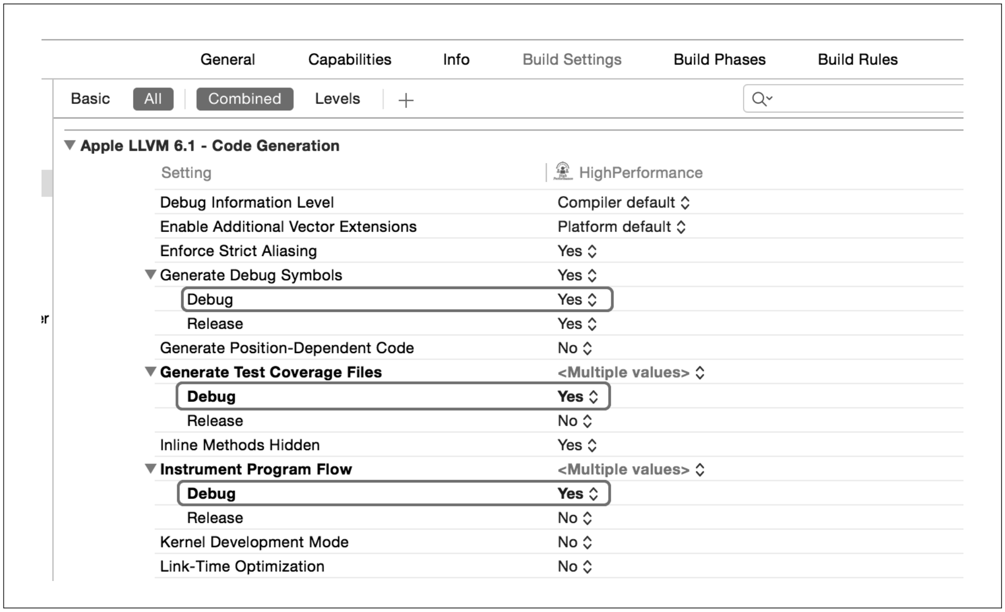


图 10-10：Xcode 开启代码覆盖率设置

这些设置会在工程的衍生数据文件夹中生成 .gcno 和 .gcda 文件。.gcno 包含重建基本代码块和块对应源码的详细信息。.gcda 包含代码分支转换的计数。

下一步是使用这些文件生成可以导为 XML 或 HTML 格式的报告。

以下的两个工具非常有用。

- `lcov`
  从多个文件收集覆盖率数据到一个统一的 INFOFILE 文件。

- `genhtml`
  用 `lcov` 工具生成的 INFOFILE 文件来生成 HTML 报告。

这些工具默认没有安装在 Mac OS X 系统上，也不包含在 Xcode 命令行工具中。使用 MacPorts（https://www.macports.org）或 HomeBrew（http://brew.sh）来安装 `lcov` 软件包。

用例 10-2 中的代码在 Xcode 的 build phases 中添加一个 New Run Script Phase，从而生成报告。

例 10-2　Xcode 代码覆盖率报告的生成过程

```
lcov --directory "${OBJECT_FILE_DIR_normal}/${CURRENT_ARCH}"
    --capture
```

```
    --output-file "${PROJECT_DIR}/${PROJECT_NAME}.info"

genhtml --output-directory "${PROJECT_DIR}/${PROJECT_NAME}-coverage"
    "${PROJECT_DIR}/${PROJECT_NAME}.info"
```

构建工程后，你可以在名为 <\your_project_name>-coverage 的文件中看到覆盖率报告。如果打开主文件 index.html，你可以看到类似图 10-11 所示的报告。

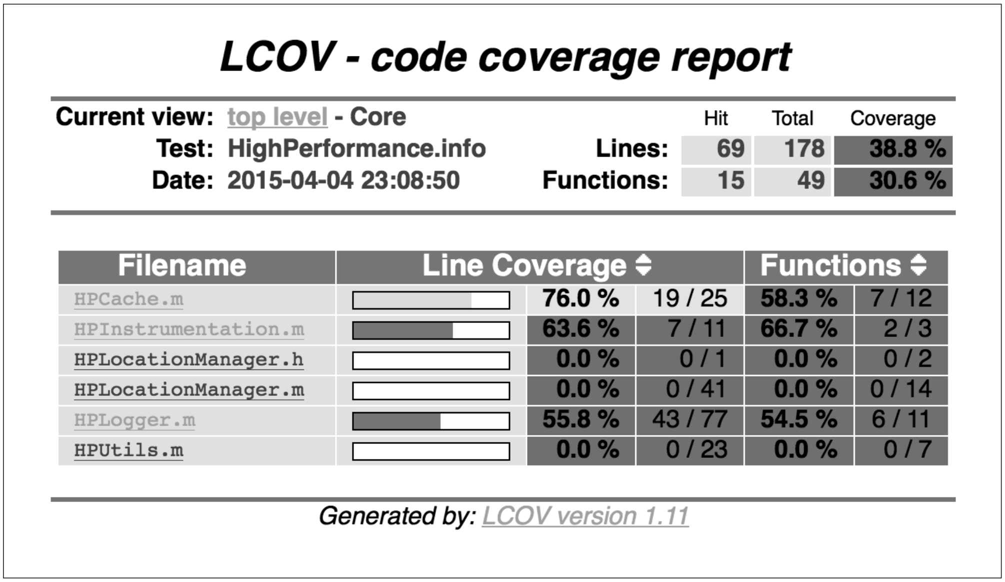


图 10-11：HTML 覆盖率报告

10.3.4　异步操作

假设我们想要测试 `HPSyncService` 类。它有执行异步网络操作的方法，也许不能立即返回服务器响应。我们需要更精确的技术来测试这样的方法。

`XCTestCase` 内置支持测试异步方法，因此，你无须使用花哨的代码来支持异步操作。

测试异步方法的步骤如下。

(1) 使用 `expectationWithDescription:` 方法来获取 `XCTestExpectation` 实例。它也在所谓的手动模式中设置 `XCTestCase`。在这种模式中，测试方法的完成并不会记为测试用例通过。
(2) 使用 `waitForExpectationsWithTimeout:handler:` 方法来等待操作完成。如果测试用例没有完成，那么将会调用回调处理的闭包块。
(3) 使用 `XCTestExpectation` 对象的方法 `fulfill` 来表示操作已经完成，等待结束。这就是第一步中提到的手动模式。

例 10-3 提供了测试异步操作的具体代码。

例 10-3　测试异步操作

```
@implementation HPSyncServiceTest

-(void)testFetchType_WithId_Completion {

    HPSyncService *svc = [HPSyncService sharedInstance];

    XCTestExpectation *expectation
        = [self expectationWithDescription:@"Test Fetch Type"]; ❶

    [svc fetchType:@"user" withId:@"id1"
        completion:^(NSDictionary *) { ❷

        //……验证数据,使用断言 ❸
        [expectation fulfill]; ❹
    }];

    [self waitForExpectationsWithTimeout:1 handler:^{ ❺
        [svc cancelAllPendingRequests]; ❻
    }];
}
@end
```

❶ 获取一个可以被满足的期望对象。我们现在处于验证测试用例的手动模式。
❷ 执行要被测试的方法，进行合理的设置和参数赋值。
❸ 如前所述，使用 XCTAssertXXX 宏来验证数据。
❹ 一旦完成，将期望对象标记为已满足。
❺ 等待期望对象被置为满足，本例等待 1 秒。
❻ 如果期望对象没有被满足，则进行清理操作。在本例中，取消任何等待的操作。

10.3.5　Xcode 6福利：性能单元测试

你可以在单元测试中进行性能测试。

XCTestCase 类提供了方法 measureBlock（http://apple.co/1SHL41A），后者可以用来测量一个代码块的性能。

例 10-4 展示了如何使用 measureBlock 来测试用例的性能。

例 10-4　单元测试中的性能

```
-(void)testObjectForKey_Performance {
    HPCache *cache = [HPCache sharedInstance];

      [self measureBlock:^{
              id obj = [cache objectForKey:@"key-does-not-exist"];
              XCTAssertNil(obj);
      }];
}
```

图 10-12 展示了例 10-4 中测试执行时的输出。

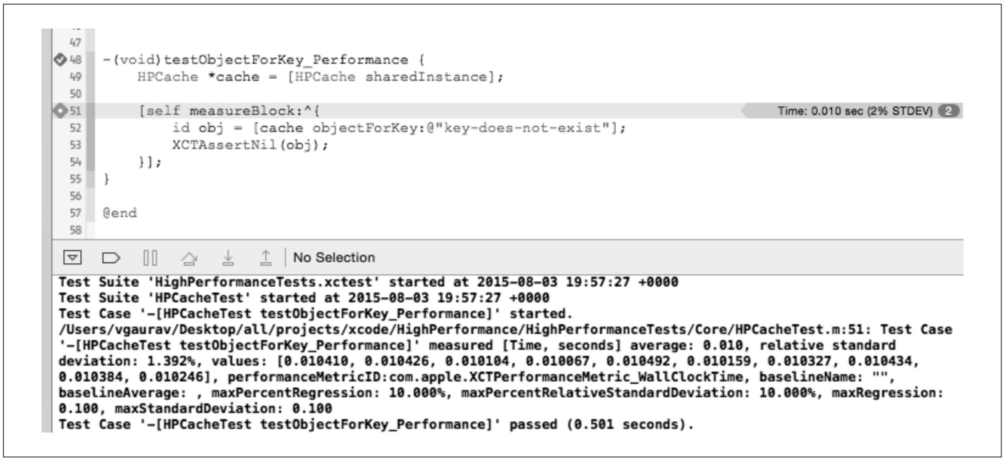


图 10-12：性能单元测试的输出结果

Xcode 同时给出运行时间的平均值及标准差。你可以为测试值偏差设置一个基线。

点击 `measureBlock` 方法所在行的对号进行设置，如图 10-13 所示。

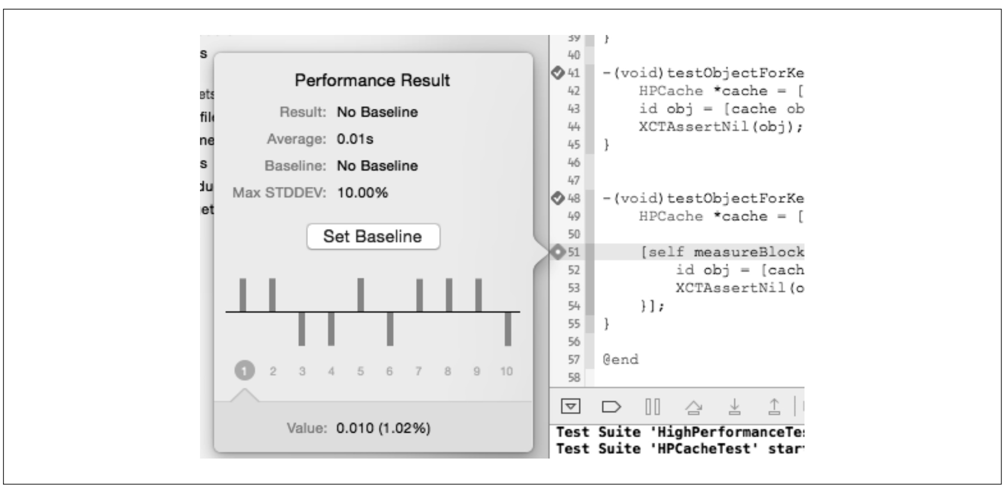


图 10-13：为性能单元测试设定基线

一旦设置好基线，输出结果将不仅显示平均值和标准差，还将显示低于基线的最差结果，如图 10-14 所示。

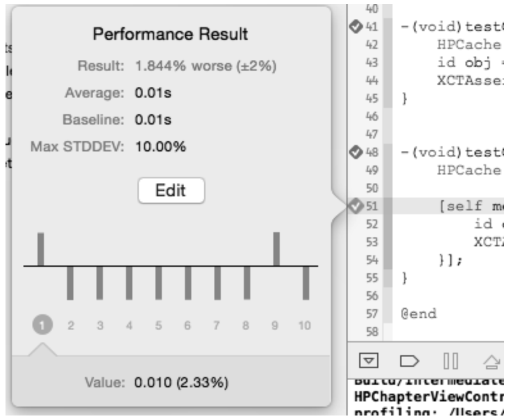


图 10-14：根据基准线测量

10.3.6　模拟依赖

之前测试的类 `HPAlbum` 是应用中最简单的类之一。它并不依赖其他子系统（如网络或持久化层）。一般来说，编写测试代码的人会遇到很多问题。

如果我们想要测试例 4-11 中的 `HPUserService`，尤其是 `userWithId:completion:` 方法，那会怎么样呢？它与 `HPSyncService` 类交互，后者中的 `fetchType:withId:completion` 从服务器获取数据。考虑以下的问题。

(1) 应用是否真的应该发起网络请求？
(2) 如果想要测试各种场景，如何告知服务器返回何种数据？
(3) 是否需要建立其他服务器来返回虚拟的数据？如果是，如何让网络层灵活可配，根据使用环境（即生产或测试）与各种服务器进行通信？
即便网络层是灵活可配的，哪怕概率很小，我们该如何保证这些配置不会带入正式发布的应用中？

也许你还有更多需要考虑的问题，这就是为什么我们需要一个系统来模拟依赖关系。带有模拟依赖的测试用例的工作方式如下。

(1) 配置依赖项以便依照提前定义好的方式运行，返回特定的值，或根据特定的输入改变至特定的状态。
(2) 执行测试用例。
(3) 重置依赖项使一切正常工作。

依赖项在 `-[setUp]` 方法中配置，在测试夹具的 `-[tearDown]` 方法中重置。

词汇

在讨论具体框架和代码前，我们先来了解一些词汇。

- dummy/double
  用于描述模拟测试对象的通用词汇，double 共有四种类型。
  - stub
    在测试期间提供封装好的数据以便被调用。它并不与应用的其他部分交互或改变至其他状态。当组件已经设计好用于依赖注入时，stub 非常实用。在测试时，配置为在特定方式下工作的依赖项可以被注入组件。
  - spy
    捕获并使参数和状态信息可用。它记录根据参数调用的方法，并帮助验证正确的方法调用。测试时，获取原始对象并创建一个 spy 对象来监控方法调用。最后，验证行为。
  - mock
    在受控制的情况下模拟一个真实对象的行为。mock 对象只为测试用例需要交互的方法进行配置。
  - fake
    除了底层实现不同，它与原始对象的工作方式一模一样。[2] 例如，模拟数据库在内存中存储数据，而且，与一般的数据库引擎不同，它可以进行快速的搜索。
- BDD
  Dan North 发明的行为驱动开发，是测试驱动开发的一种扩展。与测试驱动开发类似，行为驱动开发测试特定的功能，但也验证底层的行为。

  例如，通过测试一系列的资格证书来检查登录功能。给定一个正确的资格证书，函数应当成功，否则不通过。一个测试驱动开发的方法可以帮助你测试这种例子，但如果你想要验证这个行为，即该组件确实调用了数据库或网页服务，那就需要使用行为驱动开发。dummy 对象可以模仿或模拟底层的行为并用于验证该行为，这是行为驱动开发的关键部分。
- mocking 框架
  允许创建 dummy 的框架。至少提供了创建 mock 对象的能力，但通常被认为也可以创建 spy 对象。

OCMock（http://ocmock.org）是一个非常好的 mocking 框架，支持创建 mock 和 spy 对象。此处不对原理进行深入讨论，我们来看看使用框架的关键因素。

- 创建 mock 对象
  使用 `OCMClassMock` 宏来创建一个类的 mock 实例。
- 创建 spy 对象
  使用 `OCMPartialMock` 宏来创建一个 spy 或一个对象的部分 mock。
- stub 功能
  使用 `OCMStub` 宏对函数进行 stub 操作，实现什么也不做就返回或返回一个值。

注 2：See Martin Fowler,“Mocks Aren’t Stubs”.(http://martinfowler.com/articles/mocksArentStubs.html)

- 验证操作

  使用 `OCMVerify` 宏来验证一个底层子系统是否以特定方式进行交互（例如，一个特定方法是否以特殊的参数被调用）。

例 10-5 展示了一个使用 OCMock 框架的测试用例。

例 10-5　使用 OCMock 编写一个高级测试用例

```
#include <OCMock/OCMock.h>
#include <OCMock/NSInvocation+OCMAdditions.h> ❶

@implementation HPUserServiceTest

-(void)testUserWithId_Completion {
    id syncService = OCMClassMock([HPSyncService class]); ❷
    OCMStub([syncService sharedInstance]).andReturn(syncService); ❸

    NSString *userId = @"user-id";
    NSString *fname = @"fn-user-id",
        *lname = @"ln-user-id",
            *gender = @"gender-x";
        NSDate *dob = [NSDate date];

    data = @{
        @"id": userId,
        @"fname": fname,
        @"lname": lname,
        @"gender": gender,
        @"dateOfBirth": dob
    }; ❹

    [OCMStub([ssvc fetchType:OCMOCK_ANY
        withId:OCMOCK_ANY
        completion:OCMOCK_ANY
    ]) andDo:^(NSInvocation *invocation) { ❺

        id cb = [pinvocation getArgumentAtIndexAsObject:4];
        void (^callback)(NSDictionary *) = cb;
        callback(data); ❻
    }];

    HPUserService *svc = [HPUserService sharedInstance];
    [svc userWithId:userId completion:^(HPUser *user) { ❼
        XCTAssert(user);
        XCTAssertEqualObjects(userId, user.userId);
        XCTAssertEqualObjects(fname, user.firstName); ❽
        //……其他的状态验证
    }];

    OCMVerify([ssvc sharedInstance]);
    OCMVerify([ssvc fetchType:@"user" withId:userId completion:[OCMArg any]]); ❾
}

@end
```

❶ OCMock.h 是引入框架的主要头文件。使用 NSInvocation+OCMAdditions.h 是因为我们需要实现特定的功能。
❷ 模拟 `HPSyncService` 类的一个对象。
❸ 对类方法 `sharedInstance` 进行 stub 操作，以返回之前得到的 mock 对象。
❹ 为测试用例输入。
❺ 对实例方法 `fetchType:withId:completion:` 进行 stub 操作来满足特殊行为。
❻ 就测试用例而言，基于输入数据执行代码，我们不需要进行网络请求或执行数据库搜索，或缓存查询。
❼ 所有的配置完成后即将测试方法，此时调用 `userWithId:completion:`。
❽ 运行完成后验证状态。
❾ 验证方法以特定参数值被调用。

单元测试背后的理念是将要测试的方法视为黑盒。测试包括为其提供需要的输入，对比实际输出和期望输出，而不必了解函数的实现。这一步在例 10-5 的第 8 步中完成了，这也是*测试驱动开发*背后的理念。

第 9 步更加深入方法来测试和验证方法是否与依赖项以特定方式交互（比如，以特定参数值调用函数）。这是构成行为的部分，也是*行为驱动开发*的关键。

10.3.7　其他框架

OCMock 只是可用的框架之一，表 10-1 总结了你可以选择的其他流行框架。[3]

表10-1：iOS单元测试框架

框架类型	名称	保持器	GitHub URL
mock 对象[4]	OCMock	Erik Doernenburg	https://github.com/erikdoe/ocmock
	OCMockito	Jon Reid	https://github.com/jonreid/OCMockito
匹配器[5]	Expecta[6]	Peter Jihoon Kim	https://github.com/specta/expecta
	OCHamcrest	Jon Reid	https://github.com/hamcrest/OCHamcrest
TDD/BDD 框架	Specta	Peter Jihoon Kim	https://github.com/specta/specta
	Kiwi	Allen Ding	https://github.com/kiwi-bdd/Kiw
	Cedar	Pivotal Labs	https://github.com/pivotal/cedar
	Calabash	Xamarin	http://calaba.sh

10.4　功能测试

单元测试极大改善了对单一方法的测试，但因为对这些方法的测试都是在隔离环境中进行的，

注 3：Source: Mattt Thompson,“Unit Testing”(http://nshipster.com/unit-testing/).
注 4：用于创建模拟对象。
注 5：用于创建声明式匹配规则。
注 6：推荐以 `expect(album.name).to.equal(@"Album-1")` 替换 `XCTAssertEqualObjects(@"Album-1", album.name)`。

这要求在每个测试用例执行前进行清理工作，所以它们对整体应用的测试并没有太大帮助。

这就是功能测试出现的原因。顾名思义，功能测试确保应用的功能与预期一致。这里谈论的不是技术操作的单位，而是人为操作的单位。例如，我们不说“测试 `authenticateWithCredentials:` 方法”，而说“测试身份验证功能”，其中涉及数据输入、网络操作、UI 更新以及其他的组件交互。

功能测试更多的是在 UI 上测试，我们将应用视为黑盒。这里不是模拟对象，而是应用的实际操作。

Instruments 支持通过 UI 自动化来进行功能测试，UI 自动化是自动化的 UI 测试的简称。

10.4.1 设置

Instruments 提供一个名为 Automation 的分析模板，该模板用于创建新的功能测试或导入现有测试。

通过 Xcode → Open Developer Tool → Instruments 路径，从 Xcode 中启动 Instruments。选择 Automation 并点击 Choose 按钮（见图 10-15）。

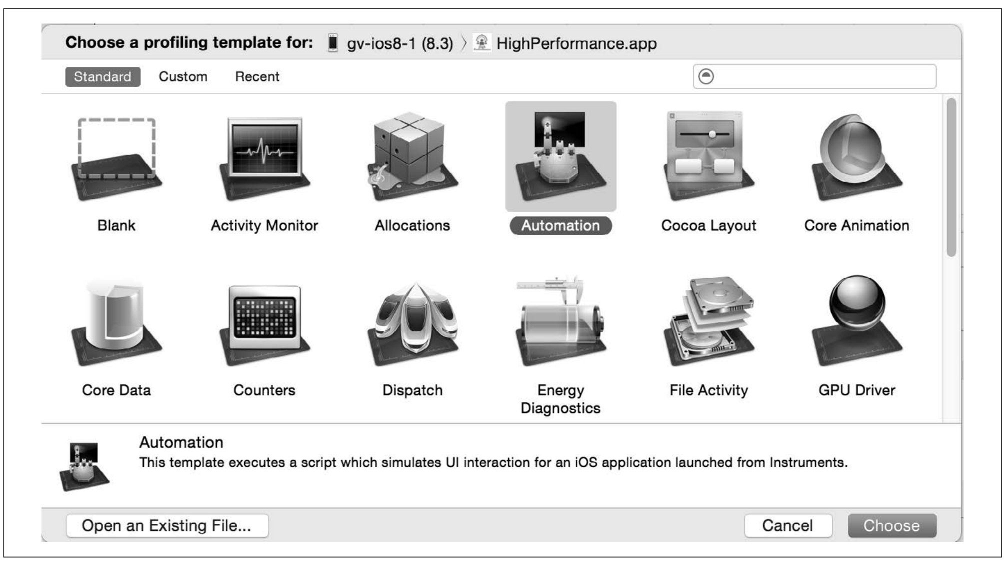


图 10-15：Instruments 中的 Automation 模板

图 10-16 展示了 UI Automation 的 Instruments 界面。你可以参考图 10-16 进行如下配置。

(1) 在设备或模拟器中打开应用。如前所述，这是运行真正的应用而不是单元测试中的模拟或独立代码。

(2) 切换到 Display Settings。

(3) 将测试重命名为其他更可读的名字。例如，如果想要测试自定义视图的组合，你可以将其命名为 `Test_CustomViews_Compos.te`。

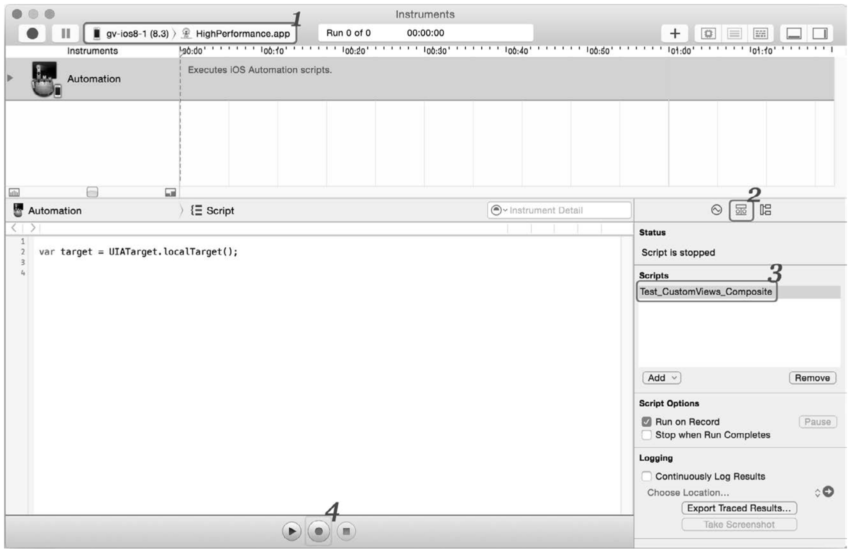


图 10-16：UI Automation——设置

下一步是在设备中启用 UI Automation。出于安全考虑，UI Automation 在设备中默认为关闭。你可以通过 Settings app → Developer → UI Automation → Enable UI Automation 来启用 UI Automation。图 10-17 展示了如何在设置中启用。

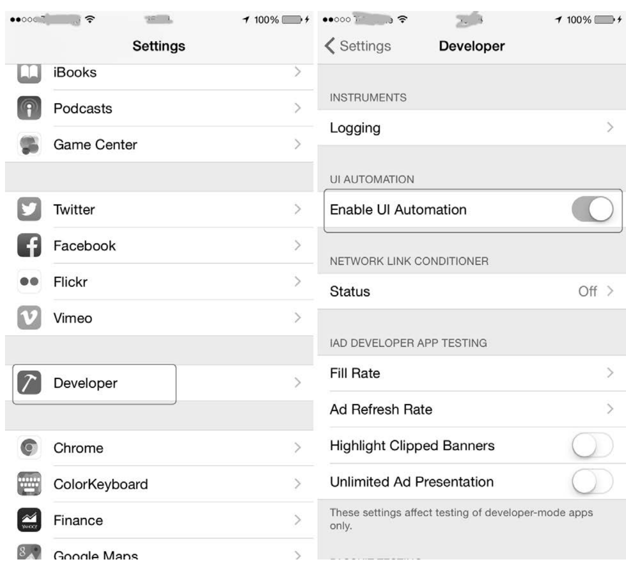


图 10-17：启用设备上的 UI Automation

完成以上步骤后，我们就可以开始编写功能测试了。

10.4.2 编写功能测试

编写功能测试的方式有两种。第一种方式是手写所有的代码。第二种方式是使用记录器生成代码，然后修改代码，我们在此使用第二种方式。

如图 10-16 的第 4 步所示，点击 Record 按钮开始记录。这会在目标设备或模拟器中启用应用。

让应用在你想要测试的场景中运行。一旦完成，点击 Stop 按钮。

尝试为以下场景创建一个测试脚本。

(1) 启动示例应用。
(2) 在 Chapters 界面部分点击 Threads。
(3) 输入 1000 表示循环的次数。
(4) 点击任意地方收起键盘。
(5) 点击 Compute Thread Creation Time 按钮。
(6) 验证结果，它应当是 "Average Creation: <\time> µsec" 的格式。
(7) 抽取并纪录创建时间。

> UI Automation 测试用例在 Javascript 中进行开发，所以你需要熟悉 Javascript。你可以在 iOS 开发者库的“iOS UI 自动化测试 Javascript 参考”中看到接口参考说明（http://apple.co/1KhVvXa）。

自动生成的代码与例 10-6 中的代码类似。

例 10-6 UI Automation——使用记录器的默认代码

```
var target = UIATarget.localTarget(); ❶

target.frontMostApp().mainWindow()
    .tableViews()[0].tapWithOptions({tapOffset:{x:0.45, y:0.62}}); ❷
target.frontMostApp().mainWindow()
    .textFields()[0].tap();
target.frontMostApp().keyboard()
    .typeString("1000"); ❸
target.tap({x:111.50, y:308.50}); ❹
target.frontMostApp().mainWindow()
    .buttons()["Compute Thread Creation Time"].tap(); ❺
```

❶ 获取目标，目标可以是设备或模拟器。
❷ 点击对应的单元格。
❸ 进入循环计数。
❹ 点击任意地方收起键盘。
❺ 点击相应的按钮。

好极了，现在我们有了正常运行的代码，可以在此基础上改进。让我们按照如下方式更新代码。

- 不要使用水平垂直坐标轴点击表格视图，我们希望可以确切点击特定的单元格。在此，我们点击第 7 个单元格。我们将使用 `UIATableView` 对象的 `cells()` 方法来获取所有的单元格，然后点击第 7 行（索引号为 6）。
- 我们需要验证结果并记录创建时间。为达到此目的，我们将使用 `UIALogger` 来记录消息并跟踪成功或失败。

在此不深入介绍这些接口，更新后的代码如例 10-7 所示。

例 10-7　UI Automation——具有确定性点击和结果的更新后代码

```
var target = UIATarget.localTarget();

target.frontMostApp().mainWindow()
    .tableViews()[0].cells()[6].tap(); ❶
target.frontMostApp().mainWindow()
    .textFields()[0].tap();
target.frontMostApp().keyboard().typeString("1000");
target.frontMostApp().mainWindow()
    .buttons()["Compute Thread Creation Time"].tap();

var msg = target.frontMostApp().mainWindow()
    .staticTexts()[0].label(); ❷

var l = msg.length;
if(msg.indexOf("Average Creation: ") != 0) { ❸
    UIALogger.logFail("Did not find average creation at the start"); ❹
} else if(msg.indexOf(" μsec") != (l - 5)) {
    UIALogger.logFail("Did not find μsec at the end"); ❹
} else {
    var t = msg.substring(18, l - 5);
    UIALogger.logMessage("Thread creation took " + t + " μsec"); ❺
    UIALogger.logPass("Hurray! Success."); ❻
}
```

❶ 点击索引值为 6 的单元格。
❷ 获取第一个静态文本的 `label`。在复杂的 UI 中，你可以使用 `accessibilityLabel` 来取代索引。
❸ 验证 `label` 的值。
❹ 如果 `label` 值不正确，则记录一条不通过的消息。
❺ 如果通过，记录计算时间以及其他内容。
❻ ……标记测试用例为通过。

关闭应用。运行测试用例。查看 Edtior Log 部分的界面（见图 10-18）。这部分会报告所有执行步骤及日志消息，包括最终失败或通过的结果。

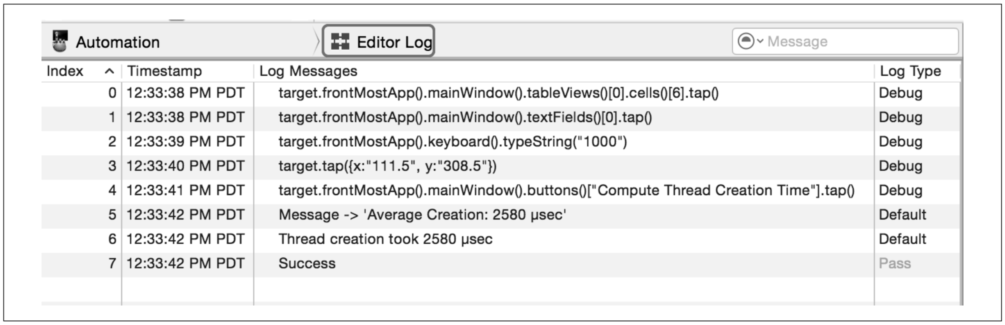


图 10-18：UI Automation——Editor Log

10.4.3　工程结构

你可能已经发现，功能测试可以使用一个庞大的 Javascript 文件或分为几个文件，每个文件表示一个特定的场景。我推荐一个场景使用一个文件。但是谨记，Instruments 一次只能执行一个文件。这意味着，测试多个场景需要多次启动应用。

管理所有测试用例的一个理想方法如下。

(1) 创建一个名为 tests 的文件夹来存储所有的测试用例。

(2) 在该文件夹中创建一个名为 allTests.js 的文件。
　该文件没有属于自己的代码，只是使用 `#import` 导入其他文件。

(3) 为场景分组创建子文件夹。
　每个场景一个文件。

(4) 从 Instruments 中调用 allTests.js 文件。

Instruments 提供一个命令行接口来执行功能性测试。这在持续集成或自动化构建管线中非常有用，我们将在 10.7 节中对此进行简单讨论。一个典型的执行命令如例 10-8 所示。

例 10-8　Instruments——命令行接口

```
$ instruments
    -t '/Applications/Xcode.app/Contents/Applications/Instruments.app/Contents/
        PlugIns/AutomationInstrument.xrplugin/Contents/Resources/
        Automation.tracetemplate' ❶
    -w '{device-uuid}' ❷
    -e UIASCRIPT '/path/to/project/tests/allTests.js' ❸
    -e UIARESULTPATH '/path/to/projet/test-results/' ❹
```

❶ `Automation` 模板的路径。
❷ 设备的 UUID 或模拟器标识符。运行 `instruments -s` 来获取模拟器列表。
❸ UI 自动化测试 Javacript 文件的路径。
❹ 测试结果所在文件夹。

10.5 隔离依赖

运行单元测试时，建议对依赖项进行分离和模拟。这样可以让测试结果免受依赖项变化的干扰。

运行功能性测试时，`OCMock` 等常用的模拟框架均不可用。在运行功能测试或性能测试时，你需要让这些子系统插件化，能够在每次测试前重置状态，隔离因网络、Core Data 等操作引起的变化。

举例说明，早前在例 4-11 中使用的 `HPSyncService` 是一个与服务器同步数据的中心。我们需要让它可配置化，以便 `shareInstance` 方法返回一个根据场景返回结果的对象。

有两种办法可以达到此目的。

- 创建一个子类，该子类返回测试场景需要的数据。之后使用方法替换或创建方法 `setSharedInstance`，从而将所有操作都转向该子类的一个对象。
  这种方法的优点在于所有操作都在测试过程中完成，一切尽在你的掌握。

- 创建一个根据特定场景返回数据的服务器。我们称之为*场景服务器*。在运行测试用例前，根据将要测试的场景配置服务器。
  这种方法的优点在于只需要对应用做最小的配置改动，改变连接的域名和 IP 地址。

  注意，服务器可以是嵌入测试过程的服务器。对于类似的嵌入 HTTP 服务器，你可以使用 `CocoaHttpServer`（https://github.com/robbiehanson/CocoaHTTPServer）。

两种方法都需要使用构建目标或 scheme 的自定义二进制包。前一种方法需要大量自定义的代码来测试不同场景，并且需要自定义 scheme 在构建时针对不同场景插入额外的代码和数据。在后一种方法中，服务器的 IP 地址需要合理配置，你可以在调试模式的构建中使用预编译宏。

另外，UI Automation 运行时并没有 API 与场景服务器交互。此时你需要更高级的框架，比如 Appium（http://appium.io）或 Calabash（http://calaba.sh）。

> Calabash 使用 Ruby 开发。如果想要使用它，你需要学习 Ruby 语言。运行 Calabash 服务器还需要一个自定义的工程目标。
> 然而，Appium 支持多种语言（https://github.com/appium/sample-code/tree/master/sample-code/examples），也需要自定义目标。它使用 WebDriver 协议与应用通信。

例 10-9 展示了根据构建设置配置场景服务器以运行测试用例的代码。

例 10-9　使用场景服务器提供场景驱动的响应数据

```
//HPSyncService.m

#define MACRO_STRING_(msg) #msg
#define MACRO_STRING(msg) MACRO_STRING_(msg) ❶

#ifndef HP_CUSTOM_REMOTE_SERVER ❷
NSString *host = @"https://my-real-server.com"; ❸
```

```
#else
NSString *host = @MACRO_STRING(HP_CUSTOM_REMOTE_SERVER); ❹
#endif

//用于Appium的someTest.js文件,采用Chai断言库和Mocha测试框架代码风格 ❺
describe("login", function() { ❻
    before(function() { ❼
        //配置WebDriver
    });

    after(function() { ❼
        //关闭WebDriver
    });

    it("should succeed with valid credentials", function() { ❽
        http.get('scenario-server.com/setup?scenario_id=valid_login'
            + '&client_id=some-unique-id'); ❾
        driver.elementByName('username').text('testuser'); ❿
        driver.elementByName('password').text('testpass');
        driver.elementByName('Login').click();

        driver.waitForElementByName('profileImage').should.be.ok; ⓫
    });
});
```

❶ 用于拼接字符的一个辅助宏。
❷ 检查是否已经定义远程服务器。
❸ 如果未定义远程服务器，则使用默认（生产环境）。
❹ 如果已定义远程服务器，则使用自定义服务器。
❺ 使用 Appium 和 JavaScript 进行功能测试。
❻ 测试套件。这个套件测试所有的登录场景。
❼ 套件的 `before` 和 `after` 方法在测试中只被调用一次，使用 `beforeEach` 和 `afterEach` 在每次测试用例前配置及在测试用例后重置。
❽ `it` 定义一个测试用例或场景。
❾ 配置场景服务器以响应特定场景，`http` 表示 HTTP 模块。服务器的 URL 仅是象征性的，你应当能理解。
❿ 设置应用的 UI，`elementByName` 方法根据 `accessibilityIdentifier` 搜索对应的 `UIView` 对象（http://apple.co/1At24WP）。
⓫ 验证 `profileImage` 是否已被载入。

例 10-9 仅仅是示范性的代码，不过它可以帮助你加快编写功能性的测试用例。

10.6 测试及组件设计

测试和测试框架可能会影响组件的设计。

例如，埋点子系统或许应该设计成单例，并抽象出几个配置步骤。但在测试时，你可能希望它可以使用一个测试应用 ID，而不是与生产环境的应用分析混为一谈。同样，对单元测

试来说，你可能需要重置所有的初始化过程，而真实的应用一次就可以完成。因此，组件需要设计为可重置的。

对同步服务等其他组件来说，你可能希望配置不同的主机地址。对每个发往服务器的请求，你可能需要附加一个场景 ID，以便服务器可以正确响应。（场景 ID 可以通过使用自定义头部来完成添加，比如，使用 `X-Test-Scenario-ID` 或修改请求本身。）

第一种方法是使用模拟数据或方法替换。但是这需要深入了解组件。而且，随着场景中越来越多的方法需要模拟数据或动态替换，这种方式会变得臃肿和难以管理。

第二种方法是让所有组件可配置。推荐你使用可重置的组件或使用依赖注入的生成器模式。

一个可重置的组件意味着，要么组件不是单例，因此可以被多次创建而没有副作用，要么它还是单例，但是通过 `sharedInstance` 方法访问，并且提供 `setSharedInstance` 及 `tearDown` 在每次测试用例完成后重置共享状态。该方法可声明在一个私有头文件中，该文件对使用此类和 SDK 的开发人员不可见。

如果组件有一些依赖项，那么组件不应该通过创建实例或使用单例来直接使用这些依赖项，而是应该提供一个自定义初始化方法或使用生成器模式，方便依赖注入。

例如，不推荐编写例 10-10 中那样的代码，你应当使用例 10-11 中那样的代码。

例 10-10　非注入依赖

```
-(instancetype) init {
    if(self = [super init]) {
        self.logger = [Logger sharedInstance];
        self.instrumentation = [Flurry sharedInstance];
    }
    return self;
}
```

例 10-11　依赖注入

```
-(instancetype) initWithLogger:(Logger *)logger
    instrumentation:(Flurry *)flurry {
    if(self = [super init]) {
        self.logger = logger;
        self.instrumentation = flurry;
    }
    return self;
}
```

如果有对其他系统的依赖，那么你可能需要创建包装器。这有助于在测试中完整模拟依赖项。它也能帮助你创建可替代的依赖。例如，与其如前文那样将 `Flurry` 作为依赖项，不如定义一个 `Instrumentation` 协议及合适的方法。在实际操作中，你可以创建一个与 Flurry 相关的 `FlurryInstrumentation` 实现。如果将来要切换到 MixPanel，你可以创建另一个 `MixPanelInstrumentation` 实现。

`Instrumentation` 协议定义如下：

```
@protocol Instrumentation <NSObject>

 -(void)logEvent:(NSString *)eventName params:(NSDictionary *)params;

@end
```

简而言之，当想让代码及应用更适合测试，让自动化测试用例胜于手工测试时，你需要提前考虑组件设计。应用及组件的设计可能会受到计划使用的测试框架的影响，因此，你需要预先谨慎选择及规划。

10.7 持续集成与自动化

持续集成可以保持代码清晰并确保构建及时更新。

一个典型的开发迭代过程（见图 10-19）涉及开发人员编写代码，然后代码会被提交到一个版本控制系统（如 Git 或 Mercurial）。每次提交会触发构建管线，接下来是所有的测试（单元、功能、性能、集成等）。功能测试也许会在模拟器或各种设备上运行。例如，当测试一个消耗大量内存和 CPU 资源的游戏时，你不能只在运行 iOS 8.x 的 iPhone 6 Plus 上测试，还要在运行 iOS 7.x 的 iPhone 4S 上测试。所有测试通过后将会产生一个内部发布的二进制包，该包用于自动化不能完成的手动测试。如果一切顺利，质量保证团队会对二进制包进行签名以发布到 App Store 商店。

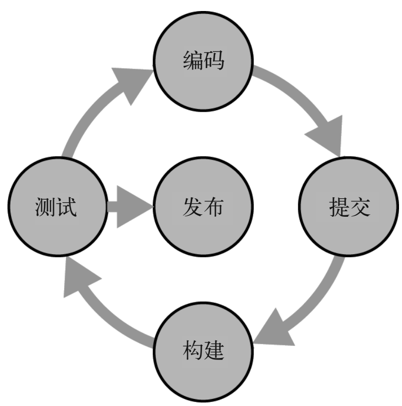


图 10-19：持续集成

除了自动化单元测试和集成测试，持续集成在构建服务器上运行，进行额外的静态和动态的代码分析、测量和分析性能、从源码创建文档，以及帮助加快手工质量保证的过程。

开源、商用以及云主机的解决方案可以帮助你实现应用的持续集成。在可选择范围内，我推荐 Travis（https://travis-ci.com）或 Jenkins（http://jenkins-ci.org）。

Travis 是一个商用解决方案，可以对 Github 上的开源项目免费试用。安装非常方便，只需添加 Travis 配置文件，声明项目的文件引用、场景以及 iOS 版本。之后将 Travis 构建引擎指向仓库。

Travis 使用 `xctool`（https://github.com/facebook/xctool）来构建项目、执行测试，并且为项目依赖集成 CocoaPods。

使用 Travis 的缺点是，它不能提供设备和操作系统组合的完整矩阵。编写本书时，它只能支持 iOS 8.1 的真实设备。

另一方面，Jenkins 是一个开源工具，一般用于任务执行管线管理，主要关注持续构建和测试软件项目，监测 cron、procmail 等外部任务的运行。构建 iOS 应用时需要 Xcode 插件。

使用 Jenkins 的优点是，整个任务管线尽在你的掌握，你可以选择使用 `instruments` 命令行工具或更高级的 `xctool`，也可以接触到运行 Jenkins 的物理设备并运行各种任务。

持续集成本身就是一个宽泛的主题，需要单独讨论。参见 John Ferguson Smart 的 *Jenkins: The Definitive Guide*（http://shop.oreilly.com/product/0636920010326.do，O'Reilly），以了解更多有关 Jenkins 的信息。

你也应该研究 Xcode Server，将其作为持续集成的方案之一。[7]

> 因为 `xcodebuild` 在项目衍生数据文件夹中生成的覆盖率和其他中间文件信息只在编译期间可用，所以需要额外的步骤与持续集成工具结合。
>
> 就 Jenkins 而言，你需要以下步骤。
>
> (1) 在 Xcode 中，添加一个 build phase，将衍生数据路径复制到一个预先声明的文件中。
>
> (2) 在 Jenkins 中，添加一个构建步骤，以便使用该文件夹生成 XML 格式的覆盖率报告。

10.8 最佳实践

正如其他章节，单元测试也有一些可遵循的最佳实践。毕竟，它的代码用来测试别的代码（别好奇又该谁来测试测试用例）。

编写单元测试时，你应当遵循以下的最佳实践。

- 测试所有的代码，包括所有的初始化方法。
- 测试参数值的所有组合。

  例如，如果一个方法接受三个参数，每个参数可以取两种值（为简单起见，假定为有效和无效），那么你应该有 2 × 2 × 2=8 种测试用例。因此，最终共有 8 种场景。

  你可以从数据库中心获取数据，或使用随机数据来覆盖所有的场景。使用人工制造或机器生成数据结合的技术被称为*模糊测试*。

- 不要测试私有方法。将被测方法视为黑盒。
- 建议消除任何的外部依赖，这保证你可以轻松驾驭各种场景。

注 7：iOS Developer Library,“About Continuous Integration in Xcode”(http://apple.co/1Kt4iXu).

- 在每个测试运行前设置状态，并在执行后清理。
  确保每次测试用例点结果不受其他测试影响。
- 每个测试用例应当是可重复的，相同的输入产生相同的结果。
- 每个测试用例必须使用断言来验证测试的代码通过与否。
- 完整的运行应当启用代码覆盖率报告。这能提供代码已被测试及未被测试的概览，以及哪个组件有较好地覆盖，哪个组件需要关注。

除了这些指导方针，别忘了运用编写代码的其他最佳实践。单元测试也是代码。

现在对功能测试也采用同样的最佳实践。唯一的区别是，与在类中测试方法不同，功能测试需要编写对业务场景，用户场景的测试用例。

此外，对功能测试而言，你还应当测试各种设备和操作系统的组合。例如，在 iPhone 5S 中的 iOS 7 系统上测试、iPhone 4S 中的 iPhone 8.3 系统上测试、iPhone 6 中的 iOS 8 系统上测试。测试的组合越多，功能测试的设备覆盖率就越高，这将确保应用可兼容各种硬件和操作系统相关的场景。

性能测试

为了追求测试代码功能性的相关因素，人们往往忘记测试与质量相关的因素，尤其是性能。

令人感慨的是，时至今日对应用性能测试的关注少之又少。大公司对性能的调研相当深入，有一系列工具和库用于测试服务器性能，但是缺失客户端的性能测试工具。如果对“iOS 应用性能测试”这个短语进行快速的网页搜索，你只能看到大量无关的广告。

绝大多数公司建立企业内的工具来测量和改进性能。个人能拥有的最好工具就是 Instruments。可以使用它对内存、CPU 以及耗电量进行分析，定位内存泄漏，等等。但要想在单元层面测试性能，你还是需要编写自定义的代码。

XCTest 提供了 `measureBlock` 方法进行块的基本性能测试。不过测试用例只会报告通过或失败以及覆盖率，并不会解释方法运行的性能因素。

即使在运行单元测试时测量性能，你也有可能得不到真实的数据，这取决于依赖项如何被模拟。

简而言之，为了测试代码的性能，你需要对希望被测试的内容编写特定的代码。

为了计算运行速度，你可以使用一个简化的计时器。例 10-12 提供了一个可用来计算耗时的计时器。该计时器支持嵌套以发现调用栈中的瓶颈。

例 10-12　跟踪运行速度的计时器

```
@interface HPTimer ❶

+(HPTimer *)startWithName:(NSString *)name;

@property (nonatomic, readonly, assign) uint64_t timeNanos;
@property (nonatomic, readonly, copy) NSString *name;
```

```
-(uint64_t)stop;
-(void)printTree;

@end

@interface HPTimer ()

@property (nonatomic, strong) HPTimer *parent;
@property (nonatomic, strong) NSMutableArray *children;
@property (nonatomic, assign) uint64_t startTime;
@property (nonatomic, assign) uint64_t stopTime;
@property (nonatomic, assign) BOOL stopped;
@property (nonatomic, copy) NSString *threadName;

@end

@implementation HPTimer

+(HPTimer *)startWithName:(NSString *)name { ❷
    NSMutableDictionary *tls = [NSThread threadDictionary];
    HPTimer *top = [tls objectForKey:@"hp-timer-top"]; ❸

    HPTimer *rv = [[HPTimer alloc] initWithParent:top name:name];
    [tls setObject:rv forKey:@"hp-timer-top"];

    rv.startTime = mach_absolute_time();
    return rv;
}

-(instancetype)initWithParent:(HPTimer *)parent
    name:(NSString *)name {
    if(self = [super init]) {
        self.parent = parent; ❹
        self.name = name;
        self.stopped = NO;
        self.children = [NSMutableArray array];
        self.threadName = [NSThread currentThread].name;
        if(parent) {
            [parent.children addObject:self];
        }
    }
    return self;
}

-(uint64_t)stop {
    self.stopTime = mach_absolute_time();
    self.stopped = YES;
    self.timeNanos = [HPUtils
        nanosUsingStart:self.startTime end:self.stopTime]; ❺

    NSMutableDictionary *tls = [NSThread threadDictionary];
    [tls setObject:self.parent forKey:@"hp-timer-top"]; ❻

    return self.timeNanos;
}
```

```
-(void)printTree {
    [self printTreeWithNode:self indent:@""];
}

+(void)printTreeWithNode:(HPTimer *)node
    indent:(NSMutableString *)indent { ❼
    if(node) {
        DDLogDebug(@"%@[%@][%@] -> %lld", indent, self.threadName,
            self.name, self.timeNanos);
        NSArray *children = node.children;
        if(children.count > 0) {
            indent = [indent stringByAppendingString:@" "];
            for(NSUInteger i = 0; i < children.count; i++) {
                [self printTreeWithNode:[children objectAtIndex:i] indent];
            }
        }
    }
}

@end

//使用率
-(void)someMethodA {
    HPTimer *timer = [HPTimer startWithName:@"method-A"]; ❽
    [obj someMethodB];
    [timer stop]; ❾
    [timer printTree]; ❿
}

//在某个其他地方
-(void)someMethodB {
    HPTimer *timer = [HPTimer startWithName:@"method-B"]; ⓫
    //做些事情
    [timer stop]; ⓬
    //或者,printTree ⓭
}
```

❶ HPTimer 类的公共 API。

❷ startWithName 创建一个新的定时器，用于计时。

❸ 计时器上下文是线程的局部对象。在示例的实现中，一旦创建计时器上下文，计时器可以在任意线程停止。这种实现可以变为让计时器对限制线程调用 stop 方法，只在创建计时器的线程中调用。

❹ 初始化——设置稍后使用的可视化层级。

❺ 使用之前创建好的辅助方法计算以纳秒为单位的时间间隔。

❻ 从线程局部存储中获取当前线程的计时器。

❼ 美化计时器树输出。

❽ 使用计时器需要调用 startWithName，为计时器赋予一个有意义的名字。

❾ 运行后调用 stop 方法。

❿ 输出运行耗时，包括任意嵌套的计时器。

⓫ 在嵌套的方法调用中，创建其他计时器。
⓬ 如上，停止计时器。
⓭ 你可以选择输出嵌套的调用树。

10.9　小结

测试应用与实现它一样重要。单元测试帮助你在最适合的层面测试，而功能测试在真实运行环境中监测应用。启用代码覆盖率是一个推荐步骤，这可以帮助你持续检查代码中哪些步骤还未被测试。

持续集成是整个发布环节中不可或缺的部分。自动化测试解放了工程师团队，尤其是与持续集成过程相结合时，这避免了应用测试中任何人工操作导致的错误。

基于测试和持续集成的背景，现在你已可以让应用开发、构建和发布过程形成一体。最小化人工干预可以减少不经意的失误，并且这解放了质量保障人员，从而他们可以去完成其他的任务。

第 11 章 工具

到目前为止，我们已经介绍了实现一个高效、高性能应用的绝大部分重要因素。本章将探讨一些用于分析和调试各种问题的工具。

在上一章中，我们学习了可以使用代码测试应用的有效性和独立功能部分的性能。但是，在分析特定任务时还需要特定的工具，这些任务包括：

- 识别和验证无障碍标签
- 从资源利用率的角度分析应用的运行时执行性能
- 分析网络和 Core Data 的使用情况
- 分析渲染性能
- 通过自动化代码执行用户交互过程
- 分析崩溃日志

可用的工具有很多，但本章将着重关注以下工具：

- 苹果公司的 Accessibility Inspector
- 苹果公司的 Xcode Instruments
- Square 公司的 PonyDebugger
- XK72 公司的 Charles

我们先看看 Accessibility Inspector。

11.1 Accessibility Inspector

为了获得更多的用户和赞誉，应用应当支持无障碍功能。此外，当地法律可能还会强制要求应用支持无障碍。例如，美国的 508 无障碍法案可能要求整个应用或应用的某些部分必

须是无障碍的。在其他情况下，目标用户可能也会决定要求，比如，旅行或医疗应用就应当支持无障碍。

除了其他属性，每一个 `UIView`（或其子类）对象可以具有 `accessibilityLabel` 和 `accessibilityHint` 属性。这两个属性控制提供给残障人士的内容。

`accessibilityLabel` 提供展示给用户的帮助文本。例如，启用 VoiceOver 可以大声读出文本内容。而在 `accessibilityLabel` 不足以满足需求时，`accessibilityHint` 提供关于 UI 元素的额外信息。

为了分析应用是否设置了正确的值，你可以使用 Accessibility Inspector。

Accessibility Inspector 可以查看应用中每一元素的无障碍信息。有两类检查器可用：一类是在 Mac OS X 系统中与 Xcode 集成的检查器，另一类则在 iOS 模拟器中使用。

若想从 Xcode 启动检查器，导航至 Xcode → Developer Tools → Accessibility Inspector。要想在模拟器中启动检查器，需要打开设置应用，通过其中的 General → Accessibility 打开 Accessibility Inspector 开关（见图 11-1）。注意，在这两种情况下，你只能测试模拟器。

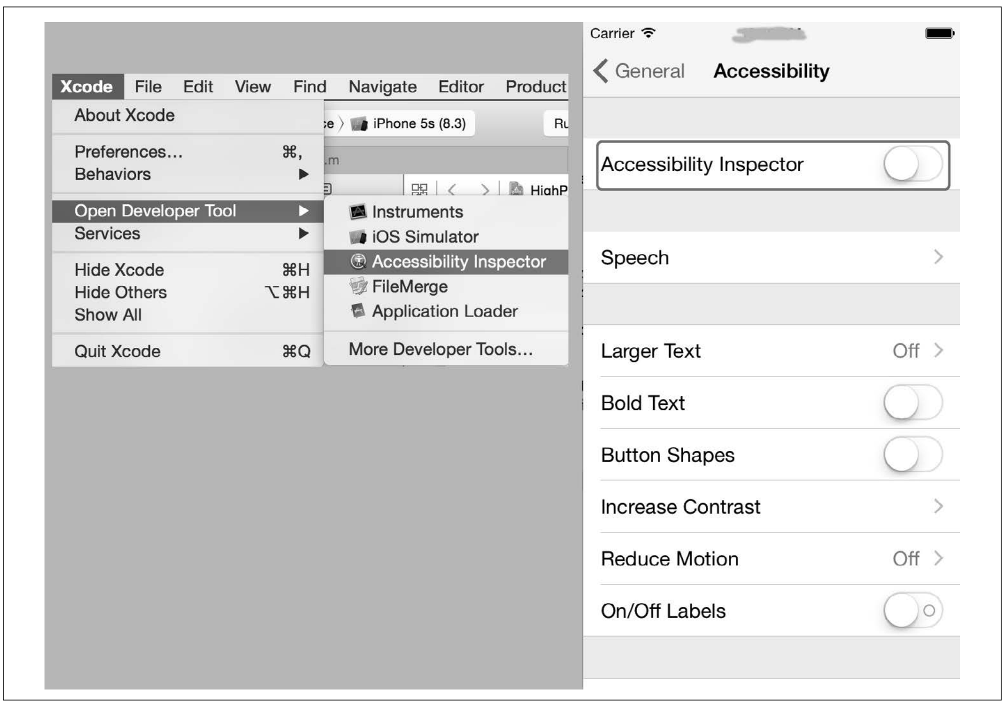


图 11-1：启动 Xcode（左）和 iOS 模拟器（右）中的 Accessibility Inspector

11.1.1　Xcode Accessibility Inspector

Xcode 中的 Accessibility Inspector 提供在 Mac OS X 上运行的应用的无障碍信息。绝大部分通用元素可以在 iOS 模拟器上正常工作，毕竟模拟器也是在 OS X 系统上原生渲染的。

图 11-2 展示了 Xcode Accessibility Inspector 的实操效果。图中高亮的元素是带有 Permissions (Tap to Request) 标签的按钮。注意，在检查器中 `accessibilityTitle`[1] 属性的值和 `UIButton` 的 `titleLabel` 属性值一致。`UILabel` 则与 `text` 属性值一致。

一个自定义的 UI 元素必须提供它自己的 `accessibilityLabel`。

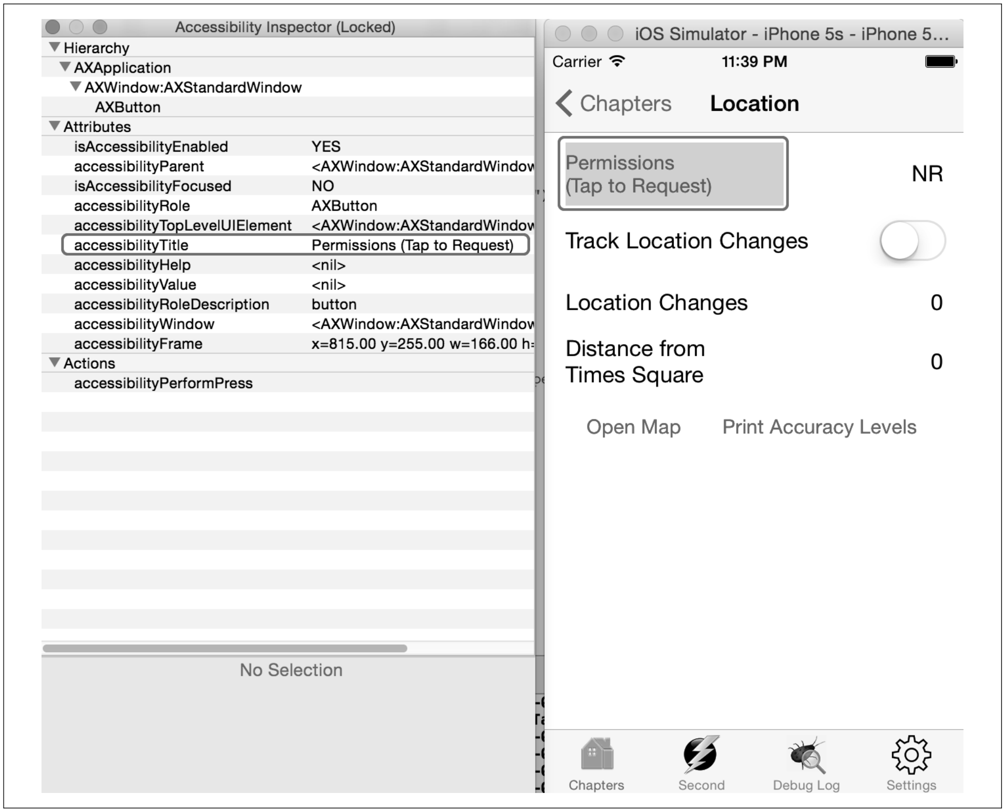


图 11-2：使用 Xcode Accessibility Inspector 检查 iOS 模拟器

11.1.2 iOS Accessibility Inspector

在 iOS 模拟器中打开 Accessibility Inspector 后，你将会看到一个浮动窗口（见图 11-3）。

注 1：工具与文档存在差异，它在 AppleKit 文档中写作 `accessibilityLabel`。详情参见 Mac 开发者库（http://apple.co/1MnmG46）。

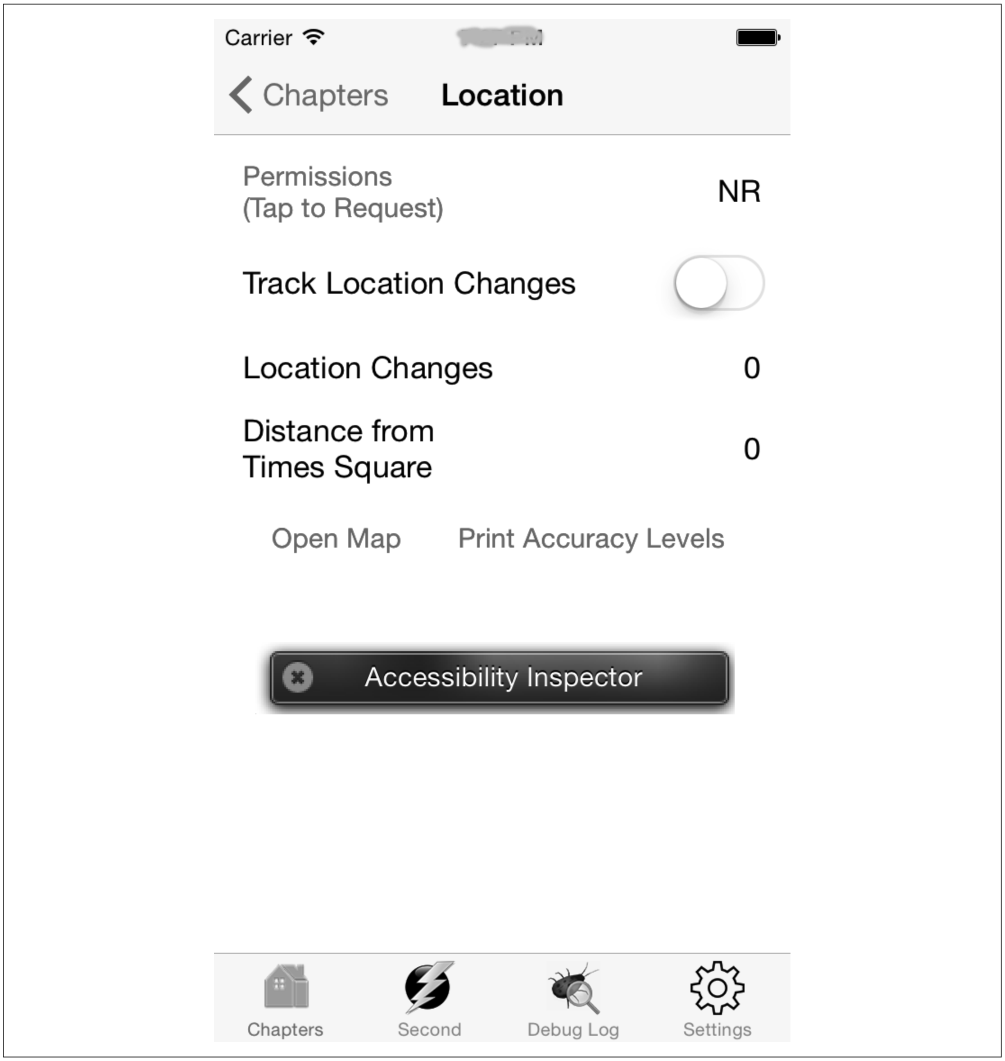


图 11-3：iOS 模拟器——Accessibility Inspector：折叠状态

如果点击 X 图标，然后选择界面上任意的控件，检查器将会展示该控件的无障碍信息。如图 11-4 所示，标记为 1 区域的按钮被点击后，检查器会展示按钮边界和 `accessibilityLabel` 值（2 区域中的 `Label`）。同时，它还会展示 `Traits` 值（http://apple.co/1HLcHl1），该值标识无障碍控件的类型以及应如何对其进行操作。可以通过 `accessibilityTraits` 属性（http://apple.co/1JrQJb6）对其进行配置。

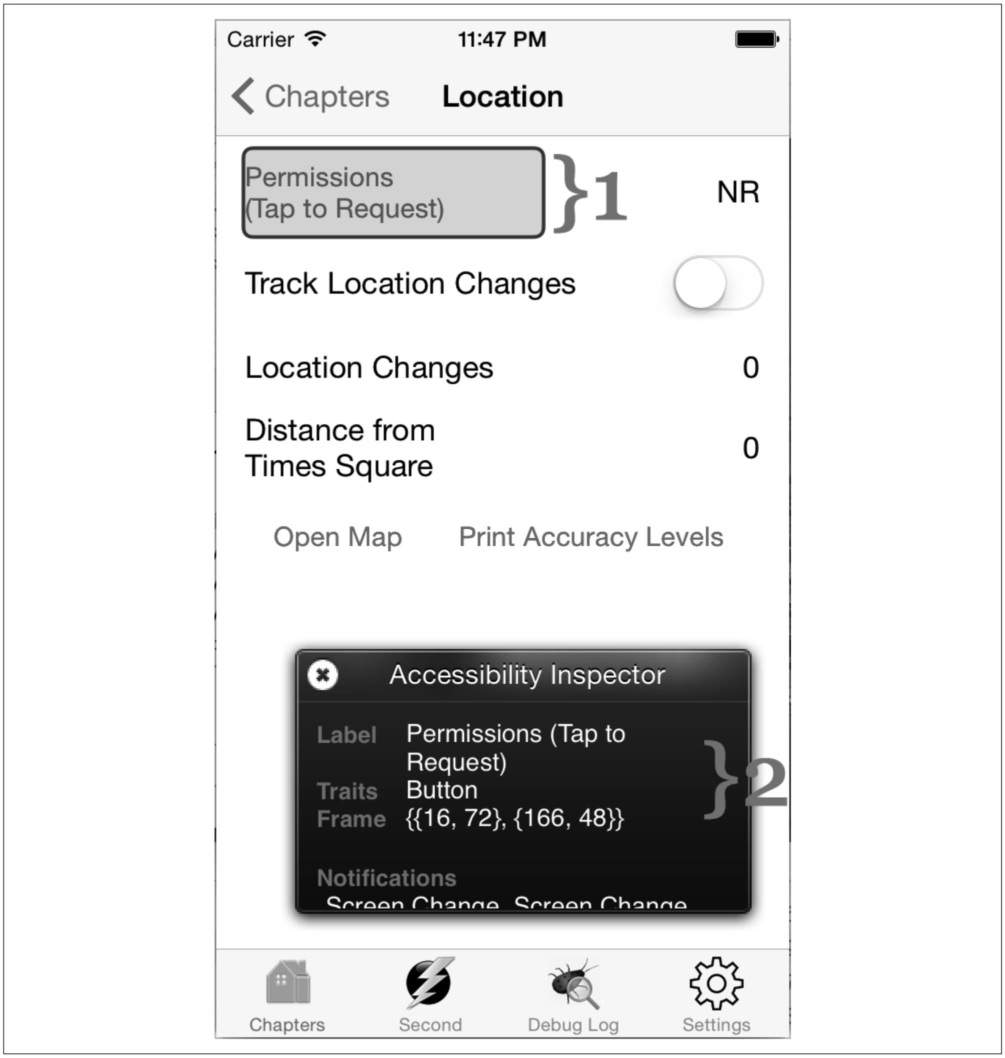


图 11-4：iOS 模拟器——Accessibility Inspector：展开状态

Accessibility Inspector 可以帮助你对应用进行快速的无障碍分析。

使用 UI Automation 可以对 UI 中每一个控件[2]对应的无障碍属性进行自动化测试。

注 2：使用 `UIAElement` 对象的 `elements` 属性，尤其是 `UIAWindow` 对象表示当前活跃应用的主窗口。详情参见 iOS 开发者库（http://apple.co/1eQy25x）。

11.2 Instruments

毋庸置疑，Instruments 是在运行时诊断、调试、分析应用的事实上的工具。在上一章探讨功能性测试时，我们简单介绍了这个工具。

本节将对 Instruments 进行深入讨论。可以通过 Xcode → Open Developer Tool → Instruments 路径打开它，如图 11-5 所示。

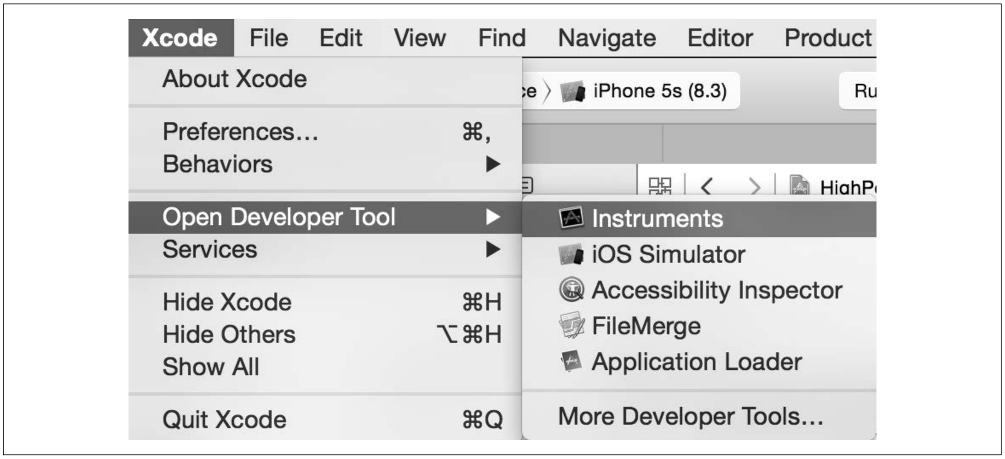


图 11-5：从 Xcode 中启动 Instruments

这将会展现一个包含各种可选模板的菜单，如图 11-6 所示。Xcode 6.3.2 中集成的 Instruments 提供了约 20 种不同的分析模板，可以从各个方面来诊断应用和设备。

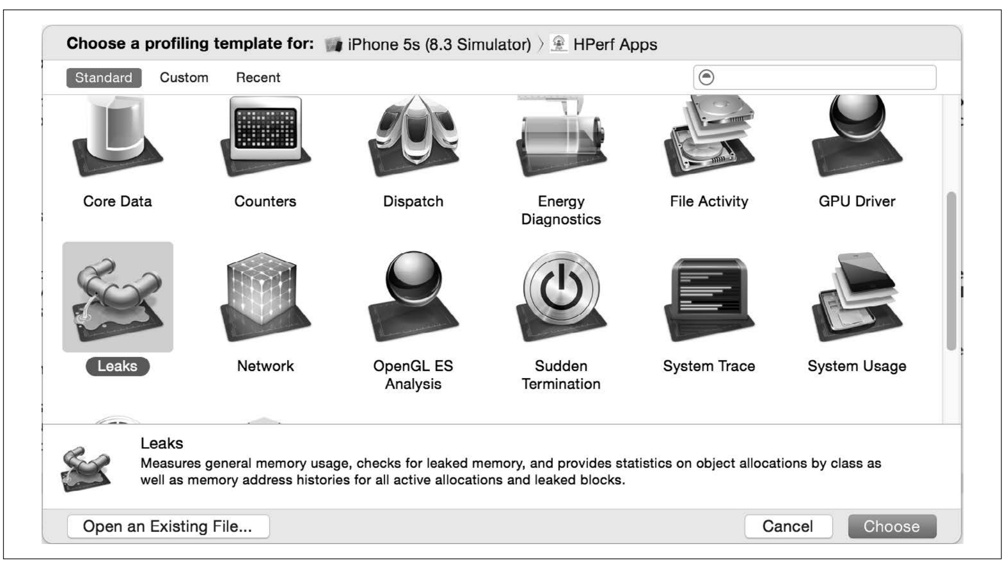


图 11-6：Instruments——模板选择界面

选择模板后按住 Alt/Option 键，你可以看到 Choose 按钮会变为 Profile 按钮，按下后即可以开始分析。

我们将深入讨论一些更有趣、更重要的模板。有关每个模板的细节，参见 iOS 开发者库中的“Instruments 用户指南”（http://apple.co/1RO7a6b）。

11.2.1　使用Instruments

打开模板后，你会发现 Instruments 窗口已经配置好相应的跟踪器。你可能希望添加更多分析项以便跟踪。每个分析项被正式称为 instrument。

很多人可能会感到困惑。工具本身名为 Instruments，而监控的分析项则被称为 instrument。例如，“CPU”是一个 instrument，“网络”也是。因此，工具被称为 Instruments 也是合情合理。

用 Instruments 分析应用、改进性能包括以下步骤。

(1) 打开一个模板。你可以使用预先定义好的模板，也可以创建一个空白的模板。
(2) 添加多个 instrument，这一步是可选的。
(3) 分析应用，这可能需要启动应用。
(4) 收集数据。
(5) 分析数据。
(6) 如有必要，更新应用。
(7) 重复上述步骤直到应用性能令人满意。

如果选择空白的模板，那么你将会看到一个空无一物的 Instruments 窗口。其他情况则会根据选择的模板预先选好对应的 instrument 项。

如图 11-7 中的主窗口所示，点击窗口中的 Library 图标将展开列表，你可以选择一个或多个 instrument 进行跟踪。图 11-8 展示了 Library 列表中的选择界面。

使用 Target 选项选择要分析的应用。点击 Record 按钮将会开始分析应用。使用 Pause/Resume 按钮可以暂停或恢复记录过程。一旦完成记录，点击 Stop 按钮。

主窗口右下方的区域称为检查器面板，用于配置记录和显示设置。

窗口左下方的区域是详情面板，用于展示窗口上半部分选中的 instrument 项的相关数据。

图 11-7 展示了 Instruments 工具的主窗口，我们可以近距离查看截屏展示的各种选项。

(1) Library 图标（打开了 instrument 列表，如图 11-8 所示）。
(2) Target 选择器（可以选择单独的应用或设备；你可以只选择在个人 Mac OS X 设备上通过开发者证书安装的应用）。
(3) 记录、暂停、恢复和停止按钮。

(4) 检查器面板。
(5) 检查器：记录设置、显示设置以及扩展详情。
(6) instrument 选择器。
(7) 按时间排列的记录图。
(8) 展示已选择分析项详情的详情面板。

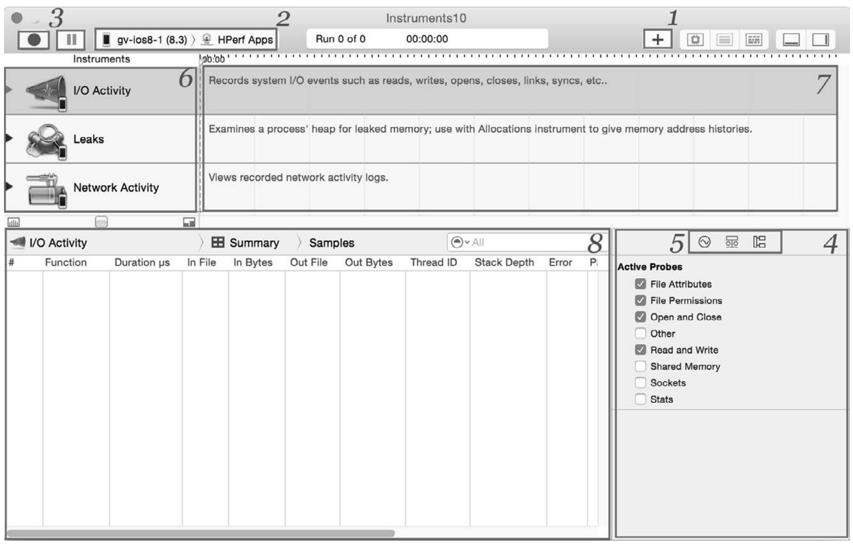


图 11-7：Instruments 主窗口

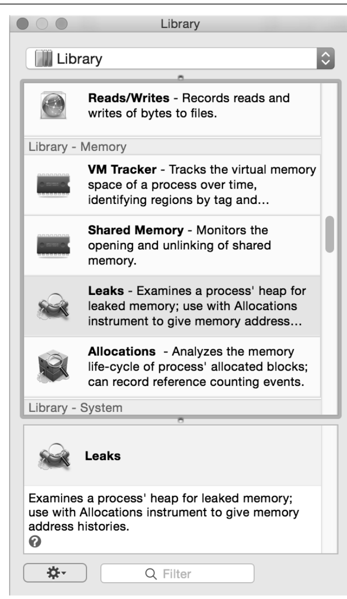


图 11-8：Instruments 库 / 选择分析项窗口

11.2.2 活动监视器

活动监视器追踪模板监控整体的设备活动（顾名思义，包括 CPU、内存、磁盘以及网络）。使用这项功能可以发现应用过度使用任何一项资源的情况。任何长期持续的过量消耗系统资源的行为尤其令人担忧。

如果发现一项资源被过度消耗，那么必须使用对应的 instrument 做进一步诊断。

该模板可用于收集一系列设备层面的统计数据。在分析开始前，在检查器面板中选中 Record Settings 项，然后选择需要收集的统计项。

图 11-9 展示了一些可用的统计项，底部的 Select statistics to list 模块展示了可以观察的所有统计项，而第二部分的 System statistics 模块则以图例显示选中的统计项。

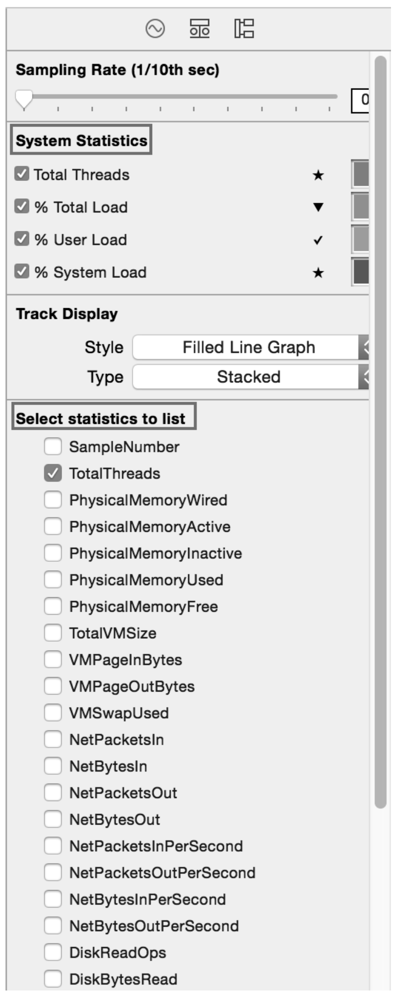


图 11-9：活动监视器统计项选择页面

分析结束后，你应该可以看到类似图 11-10 的结果。



图 11-10：Instruments——活动监视器

最上方的模块展示了统计项的用时。图 11-10 显示了图 11-9 中选中的线程数、用户负载百分比、负载总量占比和系统负载百分比。

因为活动监视器是在设备层面运行的，所以详情面板中显示的饼状图和线状图数据都是快照。它展示了四个图表。

- % CPU
  消耗 CPU 资源最多的五个应用。

- CPU Time
  运行时占用 CPU 时间最多的应用。这部分通常没什么用，因为系统应用总是位居榜首。

- Real Memory Usage（饼状图）
  占用内存最多的五个应用。你不会希望自己的应用长期出现在这个列表中。

- Real Memory Usage（线状图）
  同样展示了占用内存最多的五个应用，但以线状图展示。

11.2.3　内存分配

内存分配追踪模板可以发现应用中被遗弃的内存。被遗弃的内存指的是已经分配但不再使

用的内存空间；这部分空间仍然可以回收。

从技术角度来看，被遗弃的内存仍然是有效的，因为它被应用的某个部分引用，只不过实际上从未使用。这本质上也是内存泄漏，你不会希望在发布的应用中出现这种情况。

分配了内存空间却从未使用，这在内存消耗不高时似乎不算是很严重的问题。然而，如果这样的分配重复出现，那么内存会被不断消耗，严重时可能会导致内存不足，应用崩溃。

这样的例子包括为特征对象分配内存但从未使用的未完成特征。这只是一个简单的例子，在现实中发现类似的场景其实远比这复杂得多。

遵循以下步骤来查找被遗弃的内存。

(1) 选择 Allocations 模板来分析应用。
(2) 确定要测试的初始状态。
(3) 进行操作，让应用从初始状态到另一个状态，然后返回。
例如，这些步骤可能是登录、操作应用，然后退出登录。如果是一个新闻应用，你可能会选择特定的类别、阅读文章，然后返回类别的页面。

(4) 在 Display Settings 面板中点击 Mark Generation 按钮来产生一个堆的快照（见图 11-11）。

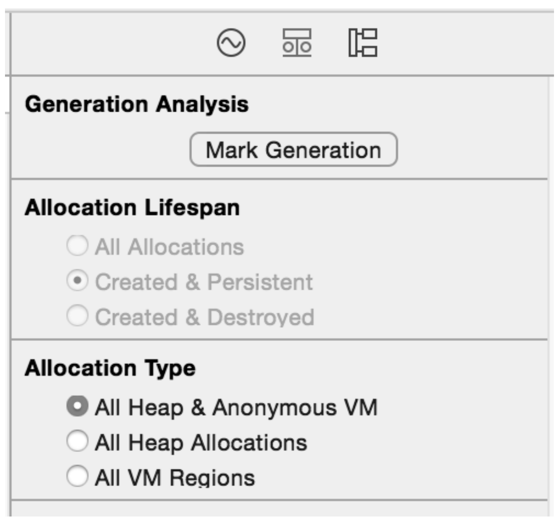


图 11-11：内存分配——Mark Generation 按钮

(5) 多次重复步骤 3 和步骤 4。
(6) 分析快照中捕获的对象，以便定位被遗弃的内存。

完成以上步骤后，你可以看到类似图 11-12 所示的数据。

根据图 11-12 中最上方的图表，内存分配 instrument 展示了内存使用率随时间的推移而稳定上升。当按下 Mark Generation 按钮时，它会创建一个版本的快照。详情面板展示了从 A 到 E 的五个版本。

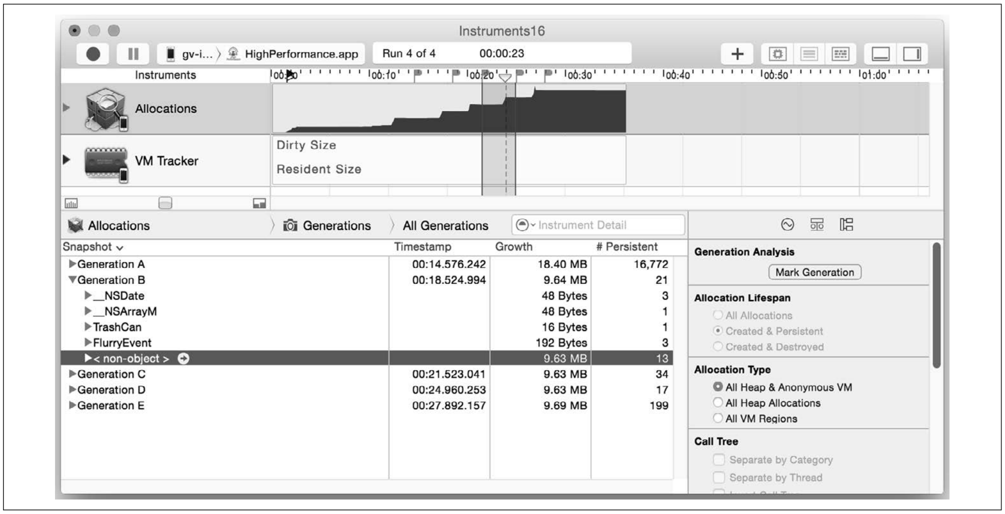


图 11-12：内存分配——摘要

在示例中，B 版本到 E 版本表明每个快照都增加了约 9.5MB 的内存。展开 B 版本的快照，我们可以发现异常的 `<non-object>` 项占用了约 99% 的增长内存。在典型的内存分析方法中，这可能很难调试，因为它的类型是未知的，我们可能永远也找不到。

点击 `<non-object>` 项的箭头来查看详情（见图 11-13）。

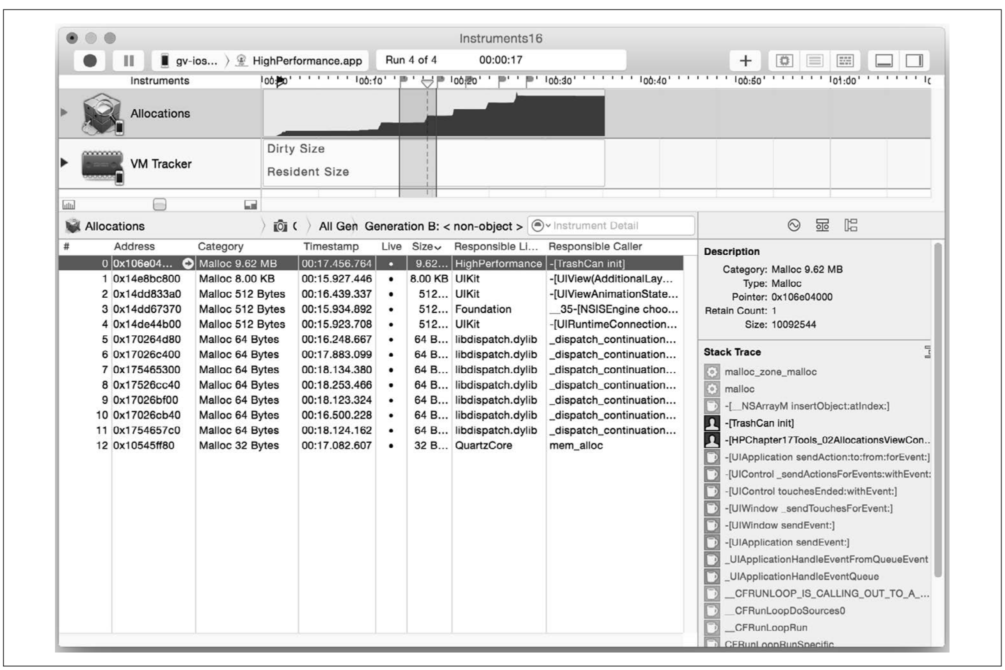


图 11-13：内存分配——详情

在所有的 `<non-object>` 内存分配中，我们感兴趣的是导致内存消耗的主要部分。在图 11-13 中，数据按空间大小降序排列。第一项表明内存消耗是在 `-[TrashCan init]` 方法中产生的，这在 Responsible Caller 一栏中体现了。在检查器面板的扩展详情界面展示了调用期间完整的栈追踪，这展现了例 11-1 中的调用树。

例 11-1　栈追踪

```
malloc_zone_malloc
malloc
-[__NSArrayM insertObject:atIndex:]
-[TrashCan init]
-[HPChapter17Tools_02AllocationsViewCon...]
-[UIApplication sendAction:to:from:forEvent:]
```

栈追踪表明应用中的某个事件调用 `HPChapter17Tools_02AllocationsViewController` 中的 `-[TrashCan init]` 方法。

这是对修复应用问题非常有帮助的信息。

11.2.4　内存泄漏

内存泄漏追踪模板可以帮助你分析内存使用情况，并检查内存泄漏。毫无疑问，要确保你的应用不会发生这样的问题。该模板还提供按类区分的对象分配的统计，以及所有主动分配和泄漏块的内存地址记录。

内存泄漏模板由内存分配和内存泄漏 instrument 组成。我们已经在上一节中研究了分配 instrument。

内存泄漏分析工具可用于发现不可达的内存引用。Objective-C 类分配的内存以类名显示，C `struct` 等其他引用对象显示为匿名实体及其分配的大小。

使用内存泄漏模板相当简单，点击后就会马上开始收集数据。

图 11-14 展示了应用在一次运行中出现几处内存泄漏的情况。

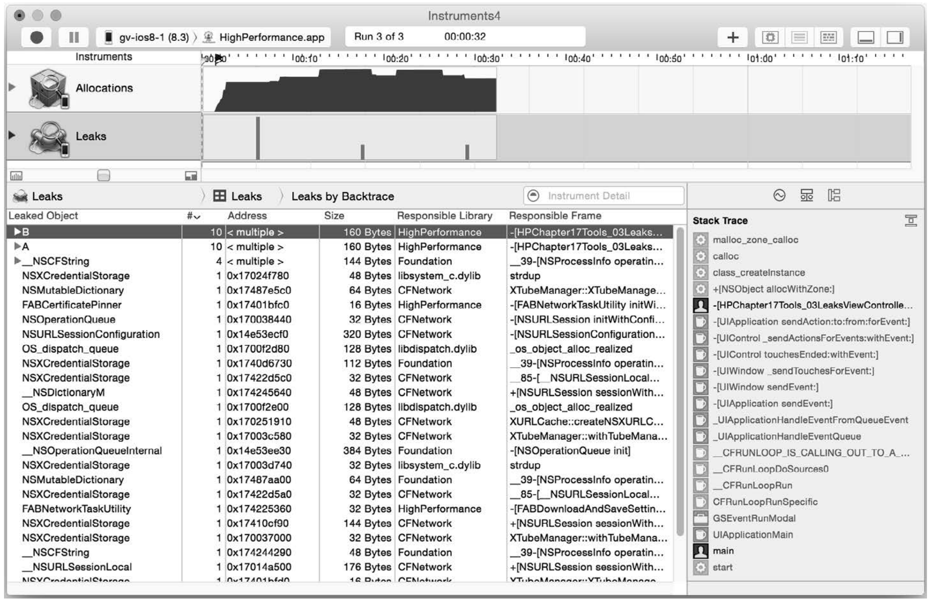


图 11-14：内存泄漏——对象和栈追踪

内存泄漏分析工具在顶部的展示栏中显示了一些红色的条，那就是发生内存泄漏的时刻。条越高，内存泄漏就越大。详情面板中列出了哪些对象发生泄漏，以及泄漏的数量和泄漏内存的大小。检查器面板中的扩展详情界面展示了当时的栈追踪，这有助于分析泄漏原因。

> 在真实的应用中，捕获内存泄漏往往是非常枯燥的过程，有时还需要几个小时的测试。
> 作为练习，你可以检查代码每次更新后是否产生了内存泄漏。深入应用的逻辑层面，对各部分进行内存泄漏分析。

11.2.5 网络

网络追踪模板帮助你分析应用产生的网络带宽连接。一旦发现过度使用网络、多域名连接、多次下载相同内容，以及使用不安全的连接等情况，则必须着手处理。

该模板由连接 instrument 组成，图 11-15 展示了使用网络追踪模版的结果。详情面板提供了调试网络使用情况时需要的内容，其中包括远程服务器地址（如果反向 DNS 查找可用，则会显示域名）、传输数据量、请求往返的平均时长和最短时间，等等。

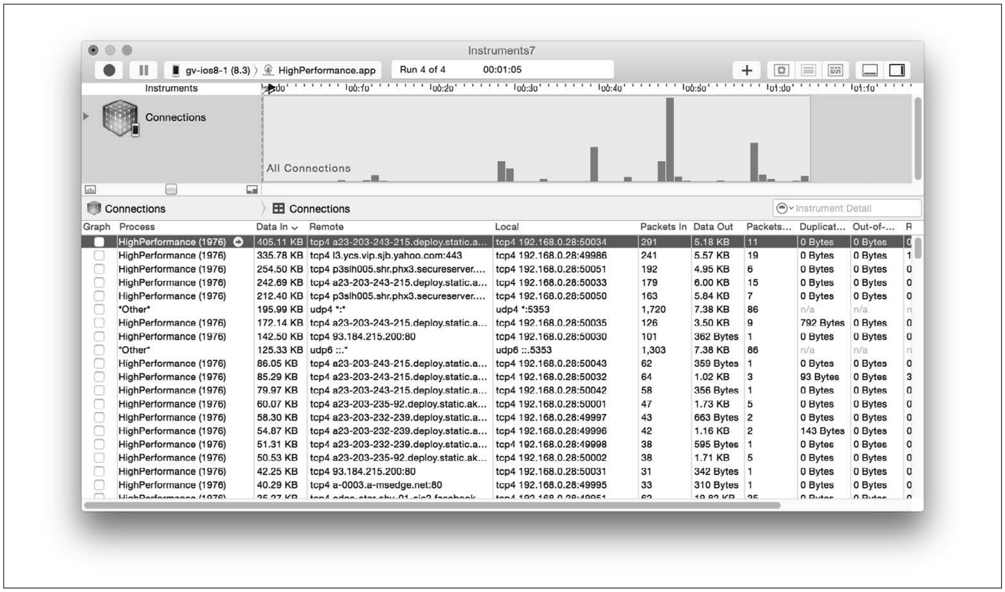


图 11-15：网络——连接

11.2.6　时间分析器

时间分析器追踪模板以固定频率收集栈追踪数据。它由同名的 instrument 组成。

在你与应用交互时，它会记录方法位于栈顶的次数。你可以通过这个数字来识别哪此方法被调用多次或者占用了大量运行时间。如果某个方法在 1000 个样本中位于栈顶 100 次，那么可以假定它占用了约 10% 的总运行时间。

如果一个方法在应用运行的大部分时间里都在执行，这可能会令人担忧。其原因可能涉及用户频繁操作或应用已经超负荷工作。根据应用的类型，这可能是一个需要关注的领域。例如，游戏应用可能需要用户频繁操作，但工具类应用则可能不需要。类似地，视频或持续的动画将会消耗大量资源，但邮件类应用则不会。

图 11-16 展示了使用时间分析器追踪模板捕获的数据。

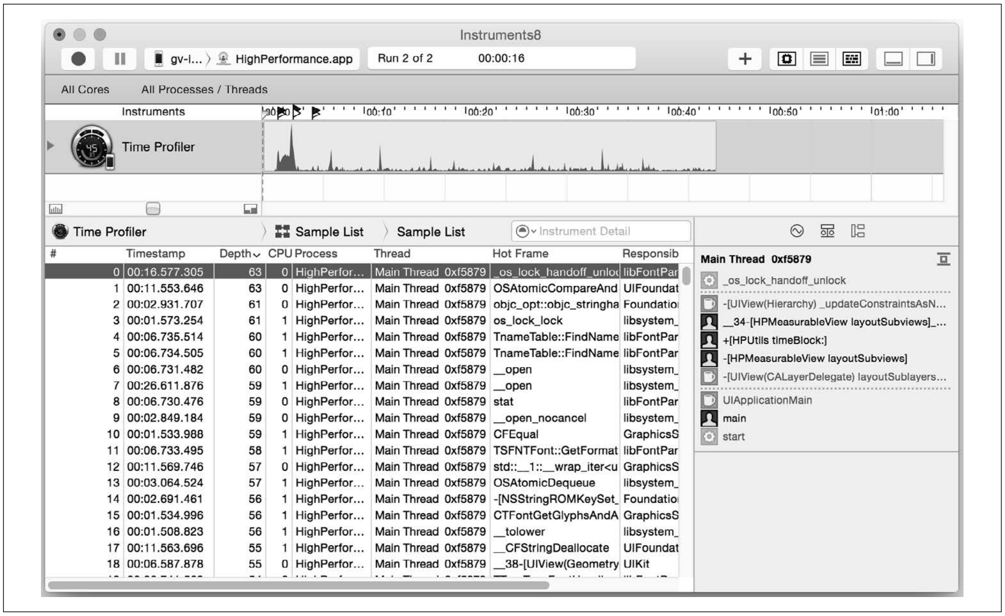


图 11-16：时间分析器

在图 11-16 中，栈追踪展示，如果应用运行缓慢，`-[HPMeasurableView layoutSubviews]` 很可能是罪魁祸首。

11.3 Xcode视图调试器

随着越来越多的功能加入应用，UI 变得越来越复杂，不仅设计和实现变得更加复杂，渲染性能也会受到影响。随着渲染的视图层级嵌套得更复杂（或层级中视图数量的增加），渲染、更新或运行动画（包括展示 `UIPickerView` 或在 `UIScrollView` 或 `UITableView` 中滑动）等行为的用时都会增加。

Xcode 的视图调试器可以帮助你在应用运行期间查看视图层级。

在 Debug 区域工具栏中点击 Debug View Hierachy 按钮可以激活视图调试器。点击后，主线程将会暂停，Xcode 中的主要视图区域就会展示当前视图层级的快照。你可以拖动快照以 3D 视角查看，如图 11-17 所示。

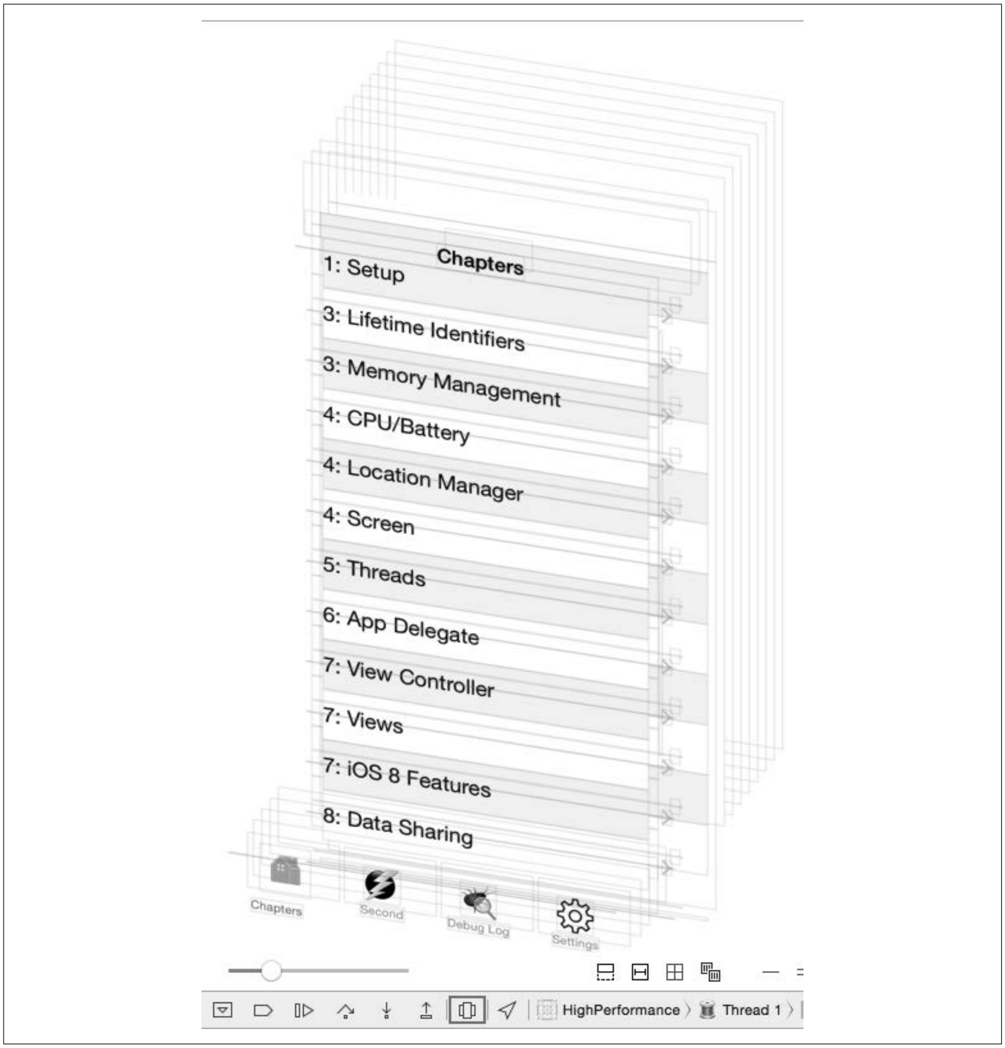


图 11-17：Xcode 的视图调试器——视图层级

如果在视图层级中点击视图元素并打开 Utilities 区，那么你应该可以查看更多关于视图的细节（见图 11-18）。

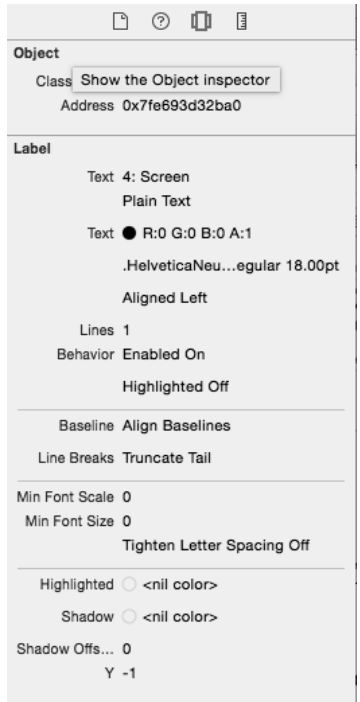


图 11-18：Xcode 的视图调试器——详细内容

11.4 PonyDebugger

使用 Xcode 视图调试器时，主线程将会暂停。因此，你需要暂停运行中的动画才能跟踪视图层级。

PonyDebugger（https://github.com/square/PonyDebugger）是由 Square 公司开发的一个远程调试工具，可以解决上述问题。此外，设备无需连接开发机器。

它由两部分组成：

- 一个网关服务器，可以在开发人员的视图和应用间建立关联；
- 一个连接到服务器的客户端。

网关服务器是用 Python 语言编写的，你可以使用例 11-2 中的命令行来安装该服务器。

例 11-2　安装 PonyDebugger 网关服务器

```
$ curl -s \
https://cloud.github.com/downloads/square/PonyDebugger/bootstrap-ponyd.py | \
  python - --ponyd-symlink=/usr/local/bin/ponyd ~/Library/PonyDebugger ❶
$ ponyd serve --listen-interface=192.168.0.1 ❷
```

❶ 下载并安装 ponyd 守护进程。

❷ 启动服务器。使用开发环境的 IP 地址，使用 `--listen-port` 参数配置端口，默认绑定 `127.0.0.1:9000`。

在浏览器中打开 http://192.168.0.1:9000 可以查看连接的设备列表。你应该可以看到类似图 11-19 的输出结果。

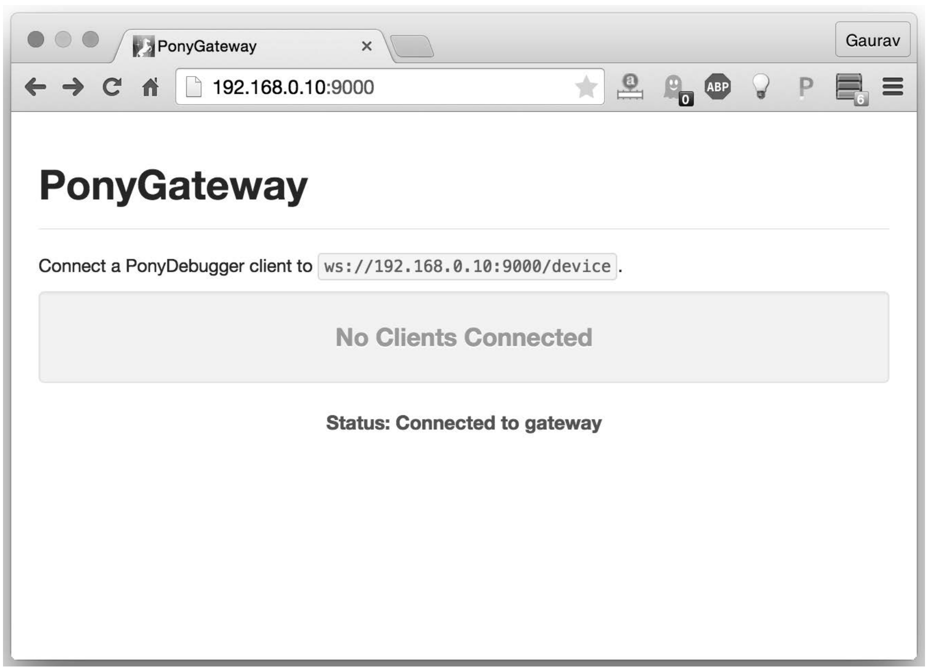


图 11-19：PonyDebugger——浏览网关服务器

客户端 API 可以通过 CocoaPods 集成到应用中。Pod 名称为 `PonyDebugger`。例 11-3 提供了集成到应用中的示例代码。

例 11-3　PonyDebugger——客户端 API

```
#import <PonyDebugger/PonyDebugger.h>

-(void)setupPonyDebugger { ❶
    PDDebugger *debugger = [PDDebugger defaultInstance]; ❷
    [debugger connectToURL:
        [NSURL URLWithString:@"ws://192.168.0.1:9000"]]; ❸

    [debugger enableRemoteLogging]; ❹
    [debugger enableNetworkTrafficDebugging]; ❺
        //[debugger forwardAllNetworkTraffic]; ❻
        [debugger enableViewHierarchyDebugging]; ❼
    [debugger enableCoreDataDebugging]; ❽
}
```

❶ 添加客户端的辅助方法。在任何网络请求发生前，从 `application:did FinishLaunchingWithOptions:` 方法中调用该方法。
❷ `PDDebugger` 是要设置的对象。
❸ 需要连接的 URL，必须是 `ws`（WebSocket 协议）。
❹ 允许日志记录在网关服务器上。使用 `PDLog` 方法代替 `NSLog` 来记录日志。
❺ 启用网络请求记录。通过替换 `NSURLConnectionDelegate` 类相关方法的机制实现。
❻ 使用该方法让 PonyDebugger 发现所有这样的类。
❼ 允许从网关服务器查看视图层级。
❽ 允许从网关服务器查看 Core Data 对象。编写本书时尚未支持更新记录。

> 因为使用 PonyDebugger 会让应用的数据暴露在外，所以建议你仅在开发调试的安装包中打开这项功能。

一旦与设备建立连接，浏览器就会展示所有允许传输的数据。

Elements 页展示视图层级。展开树形图并选中一个节点会在应用中高亮显示对应的视图。图 11-20 的左边展示了在浏览器中选中节点，右边为应用中对应的高亮视图。

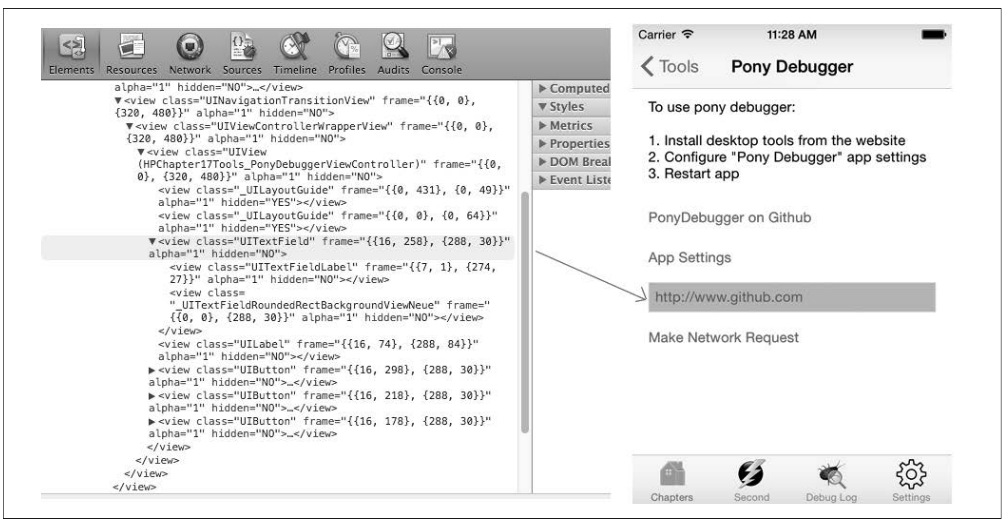


图 11-20：PonyDebugger——视图层级

在图 11-21 中可以找到网络页，其中列出了应用启动后按时间先后顺序排列的所有网络请求。记住以下重要的数据：

- 连接对应的不同域名
- 每个响应消耗的时间
- 每个请求传输的数据大小

注意，因为这个工具拦截的是 `NSURLConnection` 委托回调，所以它只能处理 `http` 和 `https`

请求。如果你使用原始套接字或其他连接方式，那 PonyDebugger 并不能拦截到它们。

在网络页上点击一个请求就能查看该请求的详细数据，与图 11-22 所示类似。你可以看到请求和响应头、响应数据，以及发送和接收的 cookie。就 POST 或 PUT 请求而言，你还可以查看请求体。

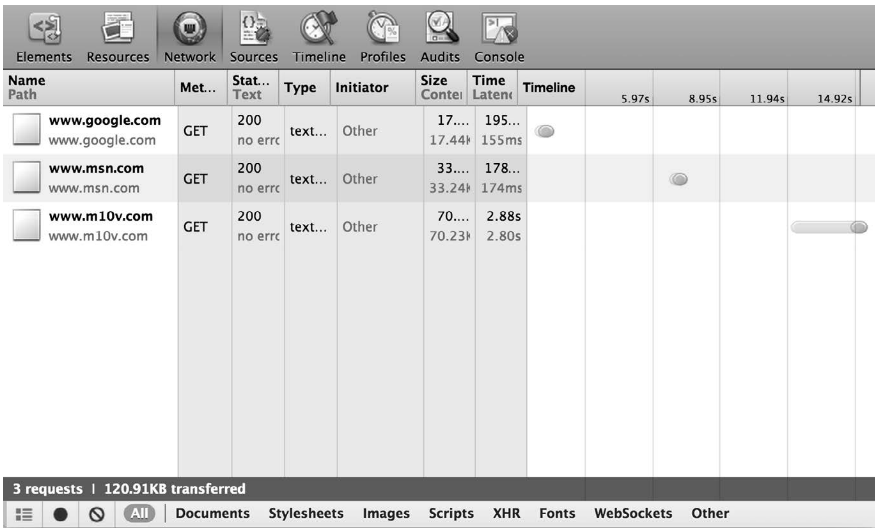


图 11-21：PonyDebugger——网络：查看所有请求

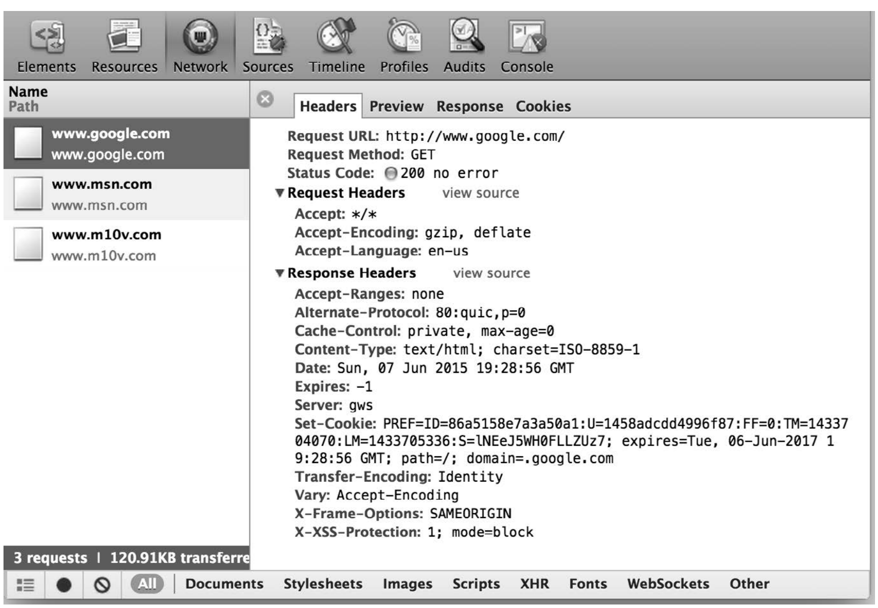


图 11-22：PonyDebugger——网络：调试单个请求

Console 页展示通过调用 `NSLog` 方法记录的所有消息（如图 11-23 所示）。`NSLog` 不仅会在调试期间将数据消息写入 Xcode 控制台，还会写到苹果日志系统中。因为需要进程间通信，所以苹果日志系统的调用开销很大。除此之外，在非调试版本的应用，包括从 App Store 安装应用时，通过 Xcode 仍然可以记录日志。记录私钥、密码等敏感信息会引发安全问题。留意日志中的此类数据。

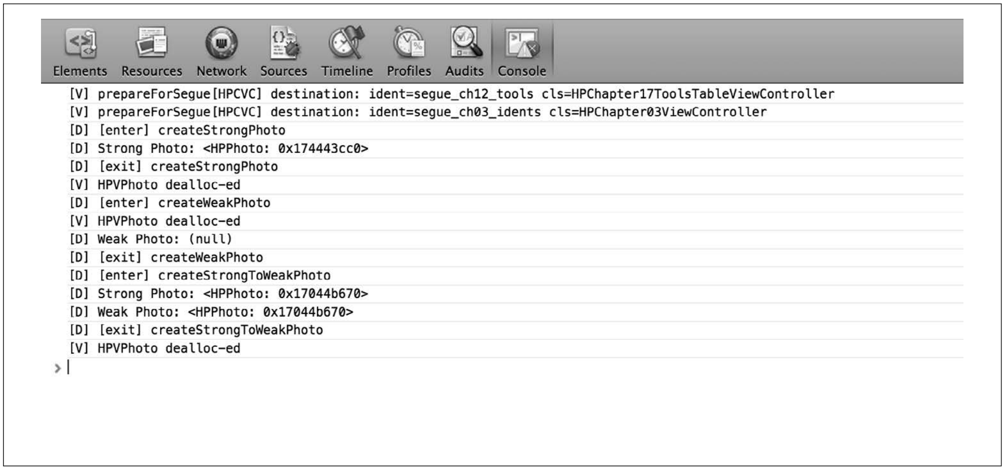


图 11-23：PonyDebugger——控制台：远程日志

使用 PonyDebugger 的网关服务器来监控和分析日志非常简单。并且，因为它是开源的，所以你可以更新代码来拦截套接字，以便分析数据以进行自动化测试。

11.5　Charles

XK72 公司的 Charles 代理服务器是一个监控、管理 `http` 和 `https` 请求的工具。

我们在 7.3.3 节中曾简单介绍过该工具，此节将继续深入探讨。

Charles 可以在被称为一个*会话*的过程中*记录*下所有的请求。因此，一个会话可以有多个请求。通过 Proxy → Start Recording 或 Proxy → Stop Recording，你可以启动或关闭记录。

每个会话可以具备一批 URL 的白名单和黑名单，可通过 Proxy → Recording Settings 的 Include 或 Exclude 页面设置这些名单，如图 11-24 所示。当白名单为空，或一个 URL 在白名单且不在黑名单中时，这个 URL 将出现在会话中。

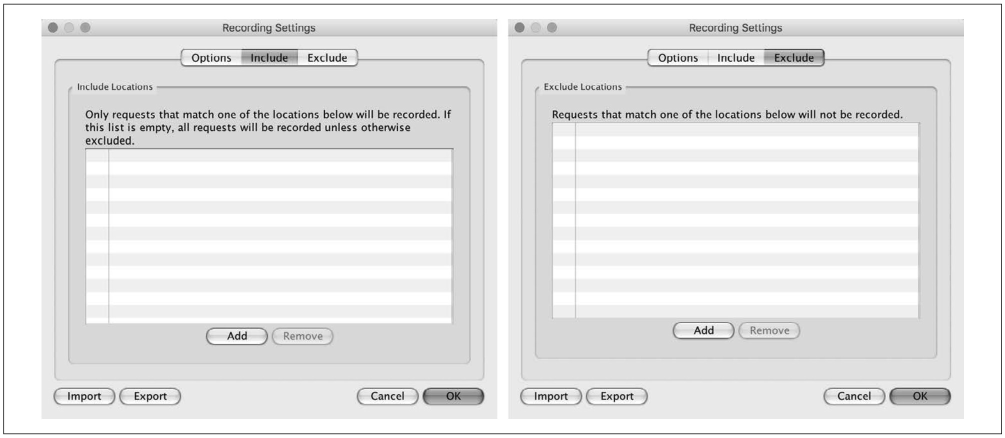


图 11-24：Charles——URL 白名单和黑名单

可以按照时间先后顺序查看会话中的 URL，如图 11-25 所示；或者根据域名为根结点、路径为子结点的属性结构查看，如图 11-26 所示。若想要检查请求的顺序，那么第一种格式很有用。若想要从某个特定的域名或路径调试请求，那么第二种格式更有利。

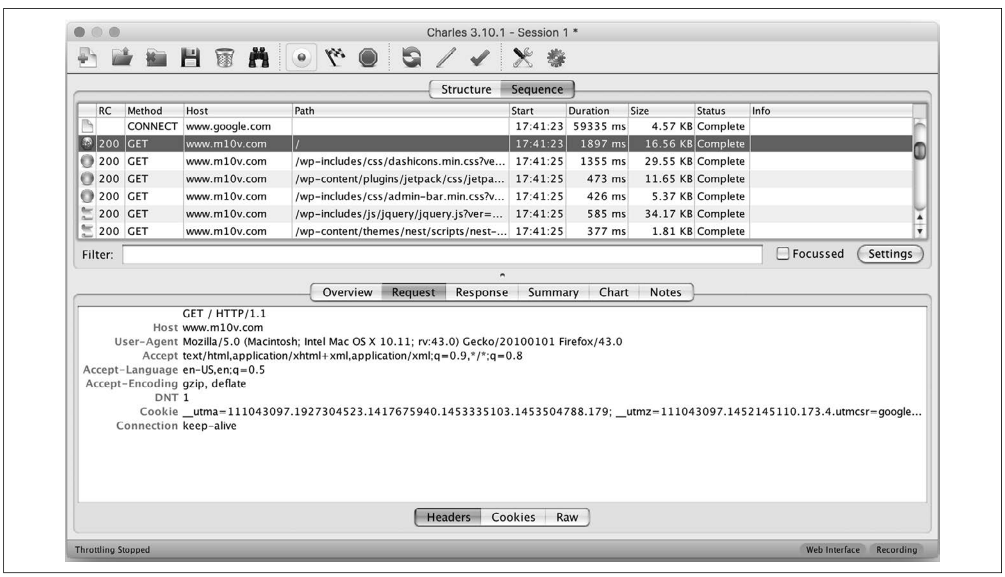


图 11-25：Charles——会话中的 URL 按时间先后顺序排列

使用树形格式可以提供一个全景概览，如图 11-26 所示。此外，该格式在以下的调试任务中非常有用。

- 在 Overview 页面可以查看一个域名及域名下的子路径相关的所有请求。你可以更深入地研究 `http` 和 `https` 请求。图 11-27 展示了以下元素：
  - ♦ 请求（请求总数、完成和失败的崩溃、服务器连接状态，以及 SSL 握手次数）

- ♦ 整体用时、DNS 解析用时、创建到主机的连接用时，以及 SSL 握手时间（如果可以的话）
- ♦ 请求（针对 `Post` 和 `Put` 方法）以及以 KBps 为单位的响应速度
- ♦ 请求及以 KB 为单位的响应大小

- 对于时间、速度和数据大小，可以查看总数、最小值、最大值以及平均值等详细数据，进而可以深入了解应用产生的网络请求。

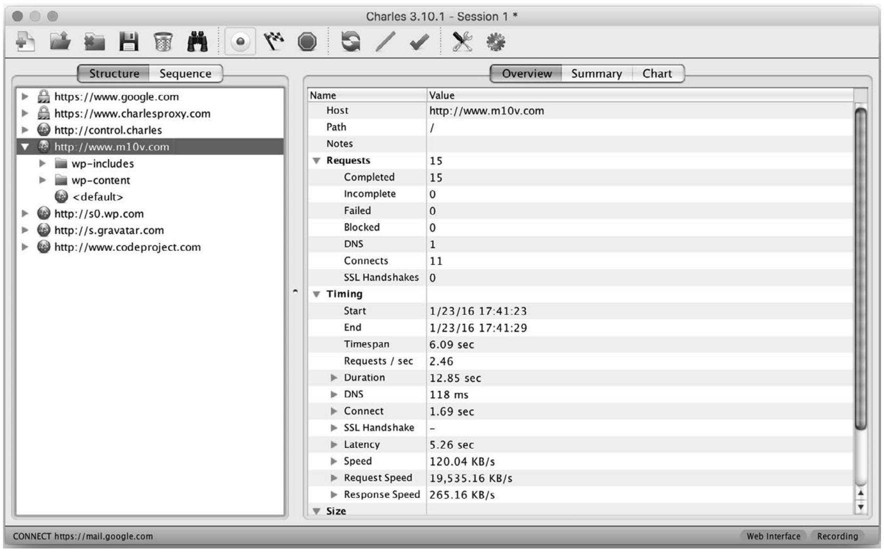


图 11-26：Charles——会话中的 URL 按树形结构排列

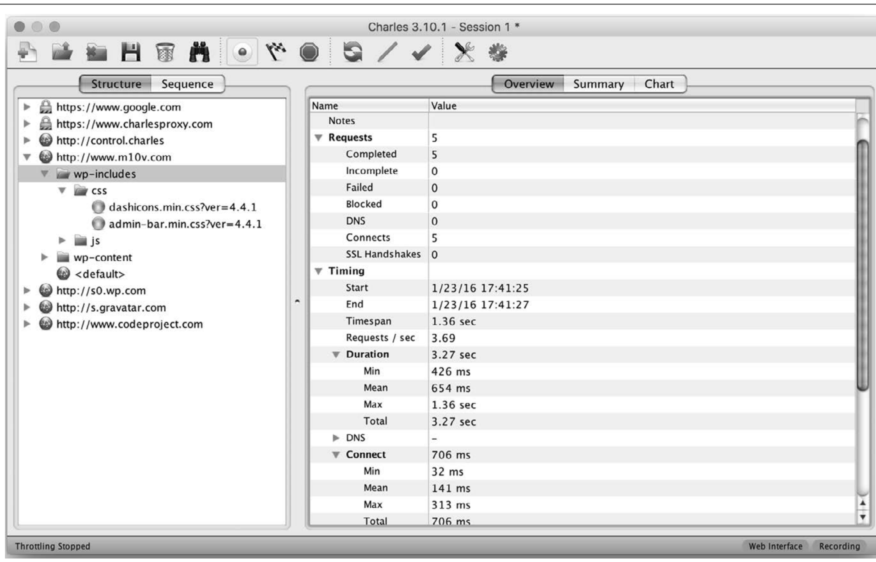


图 11-27：Charles——指定子路径的请求和响应概览

- Summary 页提供了 HTTP 的响应值、内容类型（MIME 类型）、包头大小、请求体大小，以及每个指定子路径下的 URL 的请求完成时间，如图 11-28 所示。

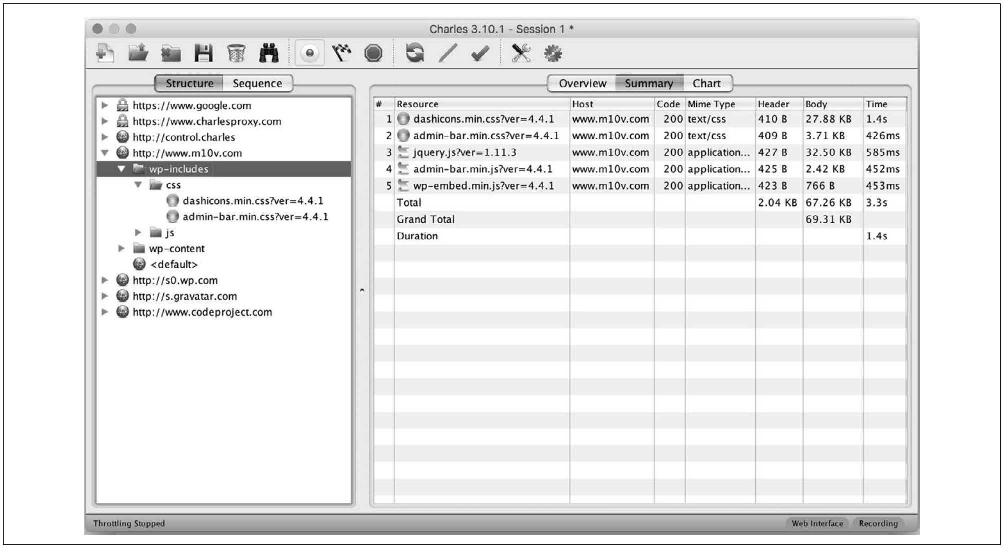


图 11-28：Charles——指定子路径下所有请求的摘要

- Chart 页提供了图形化分析来了解时间线、请求大小、持续时间和内容类型。
  (1) 当请求发起时，Timeline 子页会展示建立连接、等待响应以及接收响应（从收到第一个字节开始算起）相应的时间。这些内容体现在图 11-29 中，用于识别那些连接和响应缓慢的 URL 和域名。

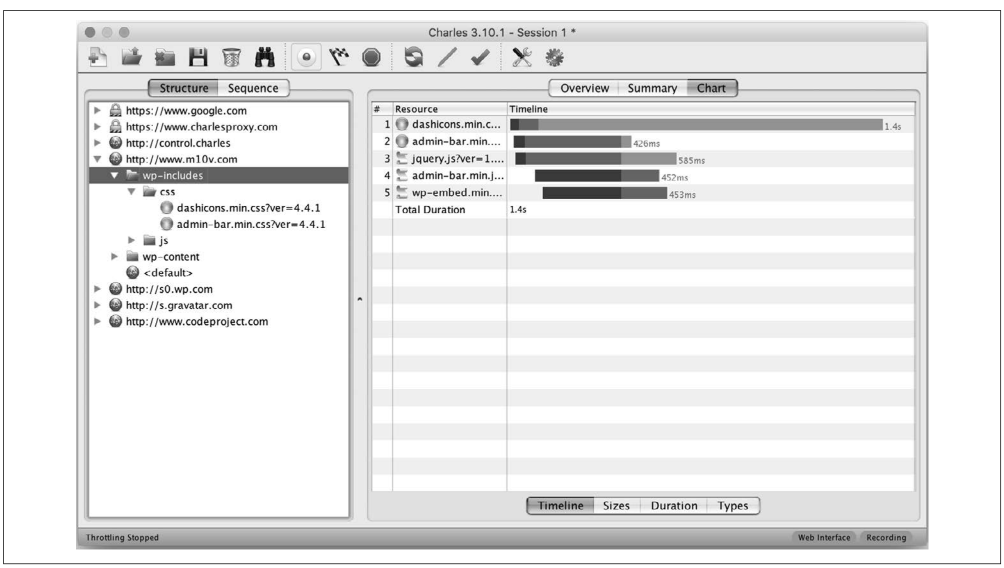


图 11-29：Charles——指定子路径下请求的相对时间轴

(2) Size 子页展示了响应大小的一个柱形图，按数据量从大到小排序（见图 11-30）。该数据可用于优化响应大小、确认服务器扩容需要，或增加缓存内容的边缘服务器。

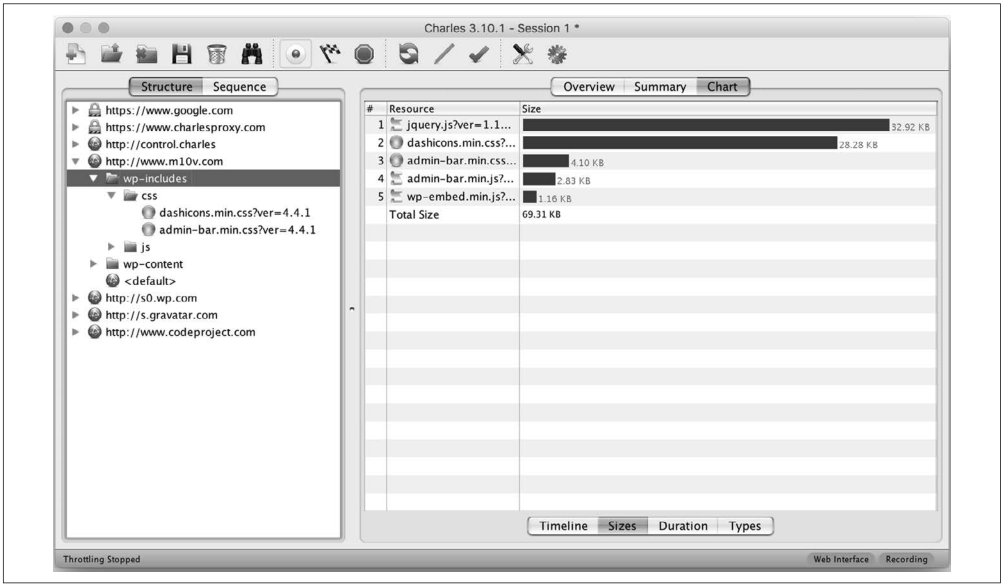


图 11-30：Charles——指定子路径下请求的响应数据的大小

(3) Duration 子页（见图 11-31）提供了每个请求的响应时间数据，依照用时长短降序排列。该数据可用于发现响应缓慢的请求和 URL，并依此优化服务器。

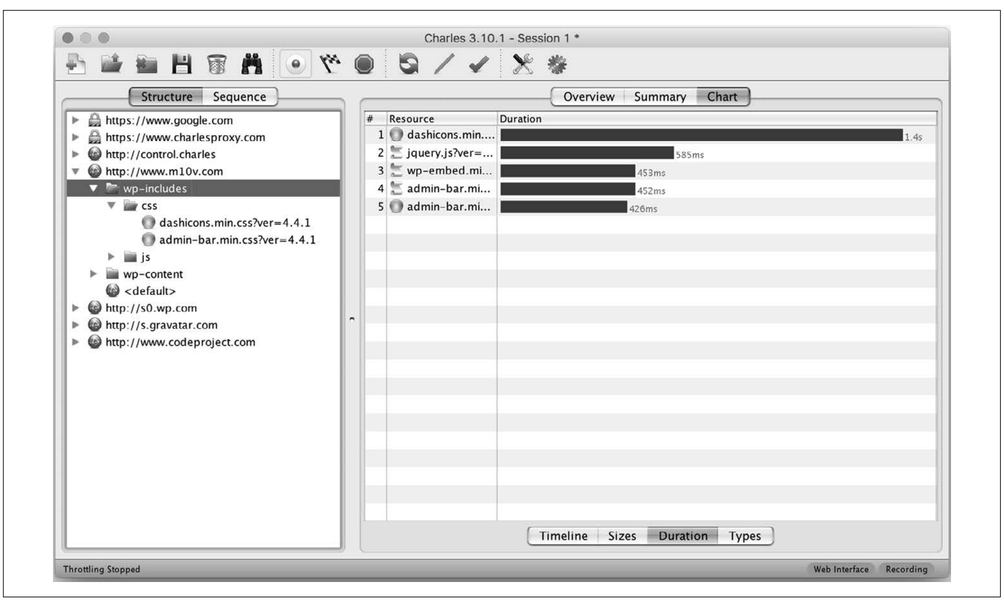


图 11-31：Charles——指定子路径下请求的响应时间

Charles 是一个从应用全面监控网络的工具。它不仅可以标识出所有发出的请求，还可以深入研究单个请求以明确任何时间和数据大小的瓶颈。

11.6 小结

调试工具可以为开发人员提供帮助。使用内嵌的代码来测量内存、CPU 使用率这样的参数，也许并不能得到真实的结果，因为这部分代码也消耗了资源。调试工具为测量性能的各方面因素提供了非侵入式的选择。

苹果公司的 Accessibility Inspector 可以测试应用无障碍方面相关的属性。

Instruments 可以监测内存使用率等性能参数、检查可能存在的内存泄漏、查看视图层级关系及其复杂度，等等。相关图表的波动都应当予以关注。同样，一个逐渐增长的内存使用图表明存在潜在的内存泄漏。

PonyDebugger 可以监测和调试视图层级，而无需暂停运行中的应用。它同样可以监控 Core Data 的使用。

Charles 可以在应用外部监控网络活动，尤其是在不侵入应用代码的情况下测试各种场景。

因为引入了许多手动操作步骤，所以使用调试工具会减慢性能的整体测试速度，但你还是应当定期使用它们来保持应用的稳定。

第12章

埋点与分析

应用开发初期的优化依据通常基于最佳实践、指南，以及从开发者机器或实验室收集的数据。然而，这些仅是数据分析的初始集合。

直到应用发布后，我们才能收集多个设备和地理位置的真实数据，帮助明确用户的使用模式和需要调整的各种场景。

在第一章中，我们探讨了用于测量和调整应用的参数，其中包括以下内容：

- 内存使用情况
- 响应时间
- 网络使用情况
- 本地存储

我们已经在前文中讨论了改善用户体验的各种策略，并明确了让应用有更高性能的特定方法，现在是时候讨论如何从真实用户那儿收集数据、分析应用使用情况、发现任何性能瓶颈、提供修复和更新版本，从而让用户对应用更加满意。

本章讨论的领域包括：如何分析收集到的生产数据，用以发现应用的使用趋势、用户行为，并通过埋点、分析和真实用户监控对应用进行改进和优化。

12.1　词汇

在展开具体内容前，我们先来了解一下本章讨论中将会用到的词汇。

- 属性
  需要获取具体值的参数，如应用版本、系统版本、位置、语言、使用内存等。

- 事件
  在应用中发生的任何行为，无论是由用户还是由应用本身触发。

  用户触发的事件包括登录、观看视频等。应用触发的事件包括冷启动、后台同步、下载邮件等。

  一个事件是一系列属性的集合，这些属性包括系统版本、设备型号、应用冷启动的用时、后台同步传输的数据量、下载邮件占用的内存，等等。

- 漏斗
  用来测量用户如何在一系列事件中切换的工具。

  一个漏斗可以用来发现使用模式以及常见任务或应用退出的位置。

- 埋点
  监控或测量性能水平和诊断错误的一种能力。在应用开发领域，它指的是向服务器发送对应的事件以便分析。

- 埋点源码
  在应用中注入埋点代码。

  注入的代码可以由开发人员手写或在编译时使用工具自动生成，你也可以使用方法替换在运行时监视。

  要想在用户的移动设备上分析生产环境中的应用，添加埋点代码是唯一的方法。你无法使用 Instruments 这样的工具，因为移动设备不能连接到开发人员的机器。另一方面，你也无法使用 PonyDebugger、Charles 这样的工具，因为移动设备可能不和工程师团队在同一个网络、被防火墙阻隔，或根本无网络连接。

- 分析
  在数据中发现和传达有意义的模式。

  在应用开发领域，分析数据来源于应用埋点。

- 同类群组
  特定时间内拥有共同特征的一组用户。

  例如，同性别的用户、在同一座城市的用户，以及在特定日期开始使用应用的用户。

- 同类群组分析
  对根据同类群组特征区分的数据进行分析。

- 归因
  将应用的销售功劳和转化功劳分配给转化路径中的接触点。

  用户可能有多个选项来开始购买或完成任务。归因模型规定了哪些选项将获得积分，并因此获得在广告活动中花费的资金份额。

- 真实用户监控
  被动监控技术，用于记录用户与应用的所有交互，将其发送到服务器，并帮助监控使用情况、趋势和出现的问题。

12.2 埋点

埋点有一些可行的技术，其中包括二进制代码重写，以及随处、即时、链接时和源码中的埋点。

我们主要关注源码层面的埋点。特别是在应用导航和资源使用方面，它可以帮助我们分析应用。

我们在第 1 章中介绍过埋点。在例 1-2 中，我们创建了 `HPInstrumentation` 类以记录事件，事件用于标识一种期望行为的发生或未发生。

例如，如果我们想跟踪*缓存*组件的性能，我们可以在它获取缓存对象的方法中记录一个缓存的命中事件或未命中的事件。

类似地，针对电商应用，我们可能想要确定用户在浏览哪个页面、哪个产品，以及为每个产品花了多长时间等信息。

从概念上来说，埋点与日志并没有区别，只是埋点的目的是将数据储存在服务器，从而达到持久化，并将其用于分析，包括离线批处理任务或实时计算。

接下来我们将了解埋点的三个阶段——规划、实现和发布。

12.2.1 规划

埋点需要深入规划。规划的第一步是就使用第三方库还是自己实现作出决定。

刚入门时，选择第三方库是比较稳妥的。随着数据、容量以及功能需求增长，你可以以后再构建自己的库。

以下是部分常用的 iOS 第三方统计库。

- Flurry（http://developer.yahoo.com/flurry）
  最受欢迎的 SDK 之一。全免费版本。支持 WatchKit。

- Mixpanel（https://mixpanel.com）
  支持更多的复杂功能，比如 A/B 测试，无需编写代码和调查。免费版本每月可以上报 25 000 个数据点。

- Appsee（https://www.appsee.com）
  支持交互热力图（如图 12-1 所示）以及录制应用使用的视频。无免费版本。

- Upsight（http://www.upsight.com/analytics）
  支持归因和同类群组分析。有免费版和付费版。

- Google Analytics（http://bit.ly/google-analytics-sdk-ios）
  从社交和电商的角度支持更多功能。每名用户每天最多有 200 000 次免费点击事件，每个会话最多有 500 次事件上报。上述限制同样适用于高级账户。



图 12-1：在一个视图控制器中的用户交互的热力图（图片来自 Appsee）

市面上还有其他的 SDK，但是上述 SDK 是很好的起步选择。

分析引擎中应当包含以下的重要功能（假设已支持通过名称记录事件，而且总是可以获取设备信息、地区、时区、位置、网络 / 运营商等基本数据）。

- 可扩展事件
  可以对任意事件添加自定义参数或内容。
- 同类群组分析
  使用可扩展事件的参数，过滤和分析其中特定的值。
- 事件计时
  可以获取事件的持续时间。

- 记录所有页面视图
  可以分析视图控制器的展示和消失。
- 用户
  可以设置用户，从而跟踪匿名用户，记录事件。
- 交易
  可以提供交易的货币价值，常用于电商或使用内购的应用。
- A/B 测试
  可以进行 A/B 测试，监控用户行为。
- 实时数据
  可以获取实时或近实时数据。根据不同的应用，可接受的事件上报延迟时间可以从几分钟到几小时。
- 安全
  可以监控安全，监控与服务器通信的安全程度，以及服务器上的数据的安全程度。
- 会话回放
  可以录制视频，之后回放。这样可以近距离监控应用的使用情况，更好地发现出错场景，更快地对其进行修复。

> 谨慎使用这项功能，因为这可能要考虑到隐私和法律方面的影响。最佳的使用场合是在私有用户研究会话。
>
> 如果你打算实现这项功能，确保使用突出的提示界面告知用户关于录制的行为。此外，在录制前还需要明确获得用户授权。

- 热力图
  可以产生热力图来确认应用的热点和盲点。
- 归因
  可以跟踪点击以及应用安装的归因。
- 活动
  支持有组织或自发的活动。常用于市场分析。
- 漏斗
  使用事件流定义漏斗。
- 原始事件
  在复杂的处理过程中使用源数据是锦上添花的一件事。这可能会在企业内部的方案中使用。

12.2.2 实现

一旦确定埋点方案，下一步就是配置。为了进行应用埋点，你需要进行如下设置。

- 确定指标

  这个过程需要产品、营销和工程师团队的合作。产品经理需要用户体验统计，营销团队感兴趣的是应用使用情况和用户感兴趣的部分，而工程团队希望了解应用的性能。

- 定义事件

  定义事件名称和相关内容来支持关键的绩效指标。

  如果工程师团队需要内存使用的均值和峰值，那么事件（我们称之为 `Heap Size`）应该包括已使用内存的数据、空闲内存（回忆例 2-40 中的代码），以及应用启动和运行的时间。

  另一个类似的例子是，如果产品经理想了解新的自动填表功能是否受到好评，那么他不仅需要使用该功能的事件，还需要了解自动填表能节约多少时间用于评价效率。

  你可能需要记时事件。例如，在一个新闻应用中，可能需要记录用户在某篇新闻或特定类别中花费了多少时间。根据使用的 SDK，你可以使用内置的计时事件。[1] 否则的话，你需要自行实现一个时间监控功能。

- 编写代码

  一旦明确所有事件，并且确定每个事件何时被调用，那就可以编写代码了。

  良好的实践经验是创建一个类，其中包含应用需要的所有埋点方法。例 12-1 包含了应用需要的一些通用方法。可以在此基础上，根据需要逐步增加。

例 12-1　埋点类——方法列表

```
@interface HPInstrumentation

+(void)logEvent:(NSString *)name params:(NSDictionary *)params; ❶

+(void)startTimerForEvent:(NSString *)name params:(NSDictionary *)params;
+(void)endTimerForEvent:(NSString *)name params:(NSDictionary *)params; ❷


+(void)logViewControllerDidAppear:(UIViewController *)vc; ❸

+(void)setLocation:(CLLocation *)location; ❹
+(void)setUserId:(NSString *)userId; ❺

+(void)logError:(NSString *)name
        message:(NSString *)message
        exception:(NSException *)e; ❻

+(void)setMinimumTimeBetweenSessions:(NSInterval)interval; ❼

@end
```

❶ 必要方法，记录一般事件。
❷ 记录计时事件。
❸ 记录用户浏览过的视图控制器。

注 1：Flurry SDK 支持计时事件。参见 Flurry 开发者文档中的“Capture Event Duration”（http://yhoo.it/1cTSpNL）。

❹ 为后续事件设置位置信息。
❺ 为后续事件设置用户 ID，常用于记录登录后的行为。
❻ 记录错误的特殊事件。
❼ 计算会话次数的特殊处理，影响日常活跃用户的计数。应当将应用切到后台再切回前台算作一次新的会话吗？或者这两个行为中间还需要一个最小间隔时间？

你可以使用 Aspects 的 CocoaPod 库（https://github.com/steipete/Aspects）来设置常用跟踪事件。比如，如果要跟踪所有 `UIViewController` 的 `viewDidAppear:` 方法，你可以使用例 12-2 中的代码。

例 12-2　使用 Aspects 的 CocoaPod 库进行无感知方法跟踪

```
[UIViewController aspect_hookSelector:@selector(viewDidAppear:)  ❶
 withOptions:AspectPositionAfter ❷
 usingBlock:^(id<AspectInfo> info, BOOL animated) { ❸
  NSDictionary *eventParams = @{
   @"ViewControllerClass": [info.instance class]
  }; ❹
  [HPInstrumentation logEvent:@"viewDidAppear"
  withParameters:eventParams]; ❺
} error:NULL];
```

❶ 添加一个钩子方法到类 `UIViewController` 的 `viewDidAppear:` 方法上。
❷ 钩子方法必须在原始方法调用后生效。
❸ 实现钩子方法。参数是 `id<AspectInfo>`，提供了调用块的上下文对象，以及原始方法（在此为 `viewDidAppear:`）需要的参数。
❹ 设置被记录事件的参数。
❺ 记录日志。

根据特殊需求还可以定义更多的方法，例如，可以定义描述购买或退款等交易过程的方法。[2]

- 验证
  记住，在发布前要先验证。测试不仅仅要考虑正确性，同时也要考虑规模。确保依赖的第三方服务有足够的容量不会被应用事件的流量压垮。

12.2.3　部署

最后一步就是部署了。它涉及发布服务器到生产（如果使用企业内的解决方案），以及发布应用到 App Store。

使用采集到的数据生成报告，明确模式和趋势并不是埋点的任务，但这在分析阶段十分重要，我们将在下文中展开讨论。

注 2：Google Analytics 的 API 支持跟踪交易事件（https://goo.gls/ibvYYt）。

12.3 分析

分析是发现和呈现有意义的数据模式，通常以数据可视化的方式呈现，比如图表、地理位置图、热力图等。

在应用开发的范畴，分析使用埋点事件产生的数据来展示有利于实现目标规划的洞见。

分析方案通常会处理一部分数据来提供高层次的趋势。百分比采样的过程可以在客户端或服务端完成。如果在客户端完成，就会是所有事件的一小部分上报给服务器。如果在服务端完成，客户端则会上传所有的数据，但服务端只处理有效事件中的一部分。

分析对发现趋势和关键指标分布非常有用。你可以用它来发现每个用户的平均会话时长或平均交易量。但不要将其用于跟踪，例如，应用已经被安装了多少次。一些特定的 API 可以准确跟踪这些统计项。

自上而下与自下而上分析

自上而下的分析系统位于客户端应用，主要监控用户行为。受监控的活动包括：用户随着内容流跳转的活动、用户正在交互的视图和所切换的视图控制器，以及在应用某个模块所花费的时间等。

自下而上的分析系统通过查看相关服务端的信息来重建用户行为和体验。例如，通过服务端接口调用来发现用户如何请求内容流的条目，并得出用户在流中进展的程度的有关结论。

这两种方法都有自己的注意事项。

自上而下的方法可能会导致网络流量的消耗增多，当连接断开或网络不可用时会丢失数据。如果使用本地持久化缓存用于批量处理，还可能会消耗本地存储空间。

自下而上的方法不会产生真实数据，它只是对用户行为的逐步重建。你可能无法得知用户花费了多长时间浏览消息流的一条内容，只是知道用户请求了哪条对应的消息。

12.4 真实用户监控

真实用户监控是监控应用以获取和分析用户的每个事务的方法。它依赖于服务端或客户端内用于监控的服务，这些服务可以监控活动的组件、其功能、应用的响应性、总体的资源使用情况和其他各种参数。

它还有各种别名，包括终端用户监控、真实用户测量、真实用户指标，等等。

12.4.1 分析与真实用户监控对比

分析同样提供这些结果。或许你在想真实用户监控到底有什么与众不同的地方。你可能会说，“它们就是对应用进行埋点，然后分析数据”。

的确，分析和真实用户监控有相同的用途，即它们都可以对应用进行埋点、分析数据，然后产出报告。

分析和真实用户监控的巨大差异在于，分析只使用一部分数据（样本）进行处理来提供高级趋势预测。

各种各样的产品提供对应用进行埋点和分析的功能，并将自己列为分析工具而不是真实用户监控工具，因为它们只记录样本。

12.4.2　使用真实用户监控

因为真实用户监控会记录所有的事件，而不仅是样本，所以你应将其用于监控关键事件。以下是一些例子。

- 任何错误，包括应用崩溃或无效状态。
- 应用新版本发布后质量的变化。
- 新功能相关的用户行为变化。
- 记录重要事务中的每个步骤。

12.5　小结

应用埋点与功能实现同样重要。它是深入了解应用质量、应用健康度、用户行为的重要途径。

使用分析工具筛选大量的数据，以便明确特征模式。你可以根据应用创建连续动作片断来了解用户行为，发现使得用户放弃整个动作的痛点或步骤。这在引导货币交易的步骤中非常有用，因为你肯定迫切想要了解这些场景下交易下降的原因。

真实用户监控应当用于监控任务的关键步骤，包括但不限于应用质量和用户行为。

再次强调，不要阻塞应用。过度使用埋点不仅会对设备上的客户端应用造成严重负担，服务端程序也需要对上报数据进行处理。

埋点不能替代调试。在开发设备上应当只使用调试工具。

第五部分

iOS 9



iOS 9 于 2015 年 9 月发布，引入了许多提高应用性能的改动。本部分将讨论这些改动，以及如何有效地使用它们。

第 13 章

iOS 9

iOS 9 包含了能够影响应用运行的一些重要改动。

除了改进标准组件以及操作系统的性能外，一些其他功能也可以提高应用性能。

本章将讨论以下内容。

- 应用的生命周期
  iOS 引入了新功能，同时，影响性能的一些原有功能被全面检修。
- 用户界面
  新加入的视图已可用，可能会提高渲染速度。全新的网页展现方式也已就绪。
- 扩展
  iOS 9 新添加了两个扩展。如果尝试实现它们，你需要仔细考虑性能影响。
- 应用瘦身
  现在不仅可以手动优化应用，你还可以使用苹果的优化工具，该工具根据应用安装的设备量身定做——期待已久的功能。

有关 iOS 9 的完整改动可参见苹果开发者网站（http://apple.co/1Bn94oT）。

13.1 应用的生命周期

iOS 9 提供了新的方式来启动并激活一个应用，同时对原有方式进行限制。

在一般场景下，*通用链接*取代了自定义 URL scheme。后者仍然是可用的，但利用该功能检测应用是否存在被严格限制了。通过这种方式入侵用户的隐私，检查用户设备中上百个应用是否存在的举动不再可行。

全新的 Spotlight 应用内搜索提供了让用户发现应用的新途径，即使应用尚未安装在设备中。这意味着拥抱新功能、编写新代码、让其顺利运行。

13.1.1　通用链接

在 iOS 8 中启动其他应用，尤其从 Web 启动，唯一的方法就是使用自定义的 URL scheme（我们在 5.2.3 节中的“深层链接”部分讨论过）。这项技术在 iOS 9 上仍然可用，但会受到严格限制。

应用仍可使用 `canOpenURL:` 方法来检测其他应用在设备上的可用性。但是，为了防止该方法被滥用，[1]iOS 9 限制了该方法的调用最多不超过 50 个唯一的自定义 URL scheme。

> 如果在 iOS 8 及更早版本上编译应用，那么设备将记录前 50 个唯一的自定义 URL scheme。而在 iOS 9 上编译则必须提供完整的清单，声明应用需要打开的自定义 URL scheme（最多不超过 50 个）。

iOS 9 引入了*通用链接*，允许应用处理之前只能被 Safari 打开的 `http` 或 `https` 链接。

通常的处理步骤如下。

(1) 源应用调用 `openURL:` 方法打开 `http` 或者 `https` 的 URL。
(2) 操作系统检测已安装的应用是否可以处理该 URL。

- 如果可以，则启动该应用。
- 否则使用 Safari 打开。

这种方法的优点在于 URL 总能被打开。换而言之，`openURL:` 方法对这些 URL 总是返回 `YES`，你不必担心需要在应用中进行分支处理。

移动网页可以使用智能应用广告条[2]引导用户下载应用。

要想使用这项功能，你需要进行如下设置。

(1) 为应用添加 `com.apple.developer.associated-domains` 授权，如图 13-1 所示。
域的取值必须依照 `applinks:{domain-to-handle}` 格式，子域不可以使用通配符。每个子域必须进行单独注册。

(2) 创建一个经过签名且名为 apple-app-site-association 的 JSON 文件，内容包含应用以及相关联的网站路径。
使用与应用签名相同的密钥。

注 1：例如，在 Twitter 上线应用图谱功能，用于收集设备上安装的应用列表后，用户隐私问题备受关注。
注 2：iOS Developer Library,“Promoting Apps with Smart App Banners”(http://apple.co/1TzDzxp).

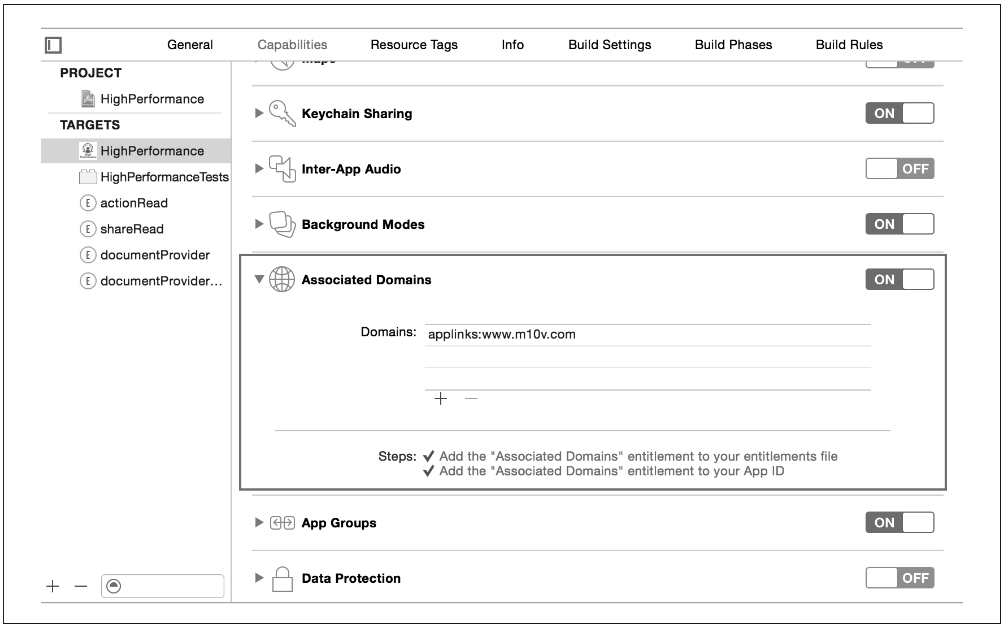


图 13-1：已关联的域名授权

这样就能在应用和域名间建立信任。

例如，对例 13-1 所示的授权 `applinks:ios.mydomain.com` 及其关联文件，应用可以处理 `http[s]://ios.mydomain.com/mypath/`、`http[s]://ios.mydomain.com/basepath/` 及其子路径。

例 13-1 通用链接：对网站关联文件进行签名

```
{
    "applinks": { ❶
        "details": { ❷
            "ABCDEFGHIJ.com.mydomain.bundleid": { ❸
                "paths": [ ❹
                    "/mypath/",
                    "/basepath/*"
                 ]
             }
        }
    }
}
```

❶ `applinks` 服务。该文件同样可用于配置 Handoff 的 `activitycontinuation` 服务。
❷ `details` 段——必须以此命名。
❸ 应用完整的密钥值，格式为 `{team-id}.{app-bundle-id}`。
❹ 路径列表，可使用通配符。

使用 `openssl` 命令行工具对该文件进行加密（见例 13-2）。

例 13-2　通用链接：签名应用网站关联文件

```
cat content.json | openssl smime
    -sign
    -inkey app-signer-private.key
    -signer app-signer-certificate.pem
    -noattr
    -nodetach
    -outform DER
    > apple-app-site-association
```

为了处理这个 URL，实现 `UIApplicationDelegate` 协议的 `application:continueUserActivity:restora tionHandler:` 方法。例 13-3 展示了处理链接的一个示例。

例 13-3　通用链接：处理链接

```
- (BOOL)application:(UIApplication *)application
    continueUserActivity:(NSUserActivity *)userActivity
    restorationHandler:(void (^)(NSArray * restorableObjects))restorationHandler {❶

   NSURL *url = userActivity.webpageURL; ❷
   //处理URL
   return YES;
}
```

❶ 应用委托回调。

❷ 使用 `webpageURL` 属性来获取 URL。

13.1.2　搜索

应用搜索[3] 提供了新方式以搜索某个应用内的公开信息，即使设备上尚未安装该应用。此类信息可被 Handoff、Siri 和 Spotlight 检索。

如果已经安装了应用，那么可以通过深层链接打开。如果未安装应用，Safari 可以引导用户到网页。通用链接确保你只需要提供一份 URL。

任何内容都可通过苹果公司的服务器索引。设备上的本地内容一开始只能在本地搜索使用，当同样的检索发生在多个设备上时，内容将被发送到苹果公司的服务器，并且建立索引以供更多设备使用。

应用搜索可以帮助用户发现未安装的应用，并在应用安装后帮助用户快速访问应用中相关的内容。若应用未安装，则用户将首先进行安装。安装完成后，用户可以直接访问相应结果，而无需在应用中实现自定义搜索。（此外，用户无需在应用中多步导航到相关页面，这样就提高了整体的用户体验。）

假设你将使用以上的一个或多个特性，那么你需要慎重考虑在应用中实现时的性能。

以下三种方式可以将内容提供给应用搜索。

- `NSUserActivity` 类中的新方法和属性可用于检索。

注 3：iOS Developer Library,“App Search Programming Guide”(http://apple.co/1RZA8RN).

- Core Spotlight 框架将应用的相关内容添加到设备上的索引，并且支持与应用深层链接。
- Web 标记让网页内容可被检索。

1. NSUserActivity

NSUserActivity 在 iOS 9 中新增了方法，可在本地索引中添加应用状态。以下的新属性加强了本地索引及公开索引。

- keywords 属性可用于链接关键词。
- eligibleForSearch 属性标识数据可用于本地搜索。
- eligibleForPublicIndexing 属性标识数据可用于在苹果服务器上的公开搜索，因此可用于跨设备检索。

例 13-4 展示了有此作用的示例代码。

例 13-4　添加到本地索引：NSUserActivity

```
NSUserActivity *activity = [[NSUserActivity alloc]
    initWithActivityType:@"com.mydomain.plist-activity-type"]; ❶

activity.title = @"iOS 9 Features"; ❷
activity.keywords = [NSSet setWithObjects:@"ios 9",
    @"new features", @"wwdc 2015", nil]; ❸
activity.userInfo = @{ ... }; ❹
activity.eligibleForSearch = YES; ❺
activity.eligibleForPublicIndexing = YES; ❻
[activity becomeCurrent]; ❼
```

❶ 根据一个已注册的活动类型创建一个活动。
❷ 设置在搜索结果中展示的标题。
❸ 与数据关联的关键词。
❹ 设置一个与活动相关联的 NSDictionary 类型数据。
❺ 设置活动可用于搜索。
❻ 设置活动可用于公开索引。数据将会发往苹果服务器并建立索引。它将会在公开的 Spotlight 与 Safari 搜索结果中出现，并且根据人气指数排名。
❼ 标识应用当前的状态（激活该活动）。一旦完成，它将被添加到通用索引（CSSearchableIndex）中。

以下是与性能相关的一些提示。

- 提供充足的关键词使得内容可被检索，但切勿滥用。注意，搜索可能是本地运行的，多个关键词不仅会降低排名，同时也会降低搜索速度。
- userInfo 可用于存储与活动相关的自定义数据。因为数据将存储在应用外，所以尽可能保持小的体积。这些数据在建立索引时会被序列化，而用户打开搜索结果时会反序列化。数据越多，耗时越长。

2. Core Spotlight

iOS 9 引入了新的 Core Spotlight[4] 框架。通过提供对应用内容的索引以及管理设备中的索

注 4：iOS Developer Library,“Core Spotlight Framework”(http://apple.co/1QFKy5I).

引，它可以让应用参与到搜索中来。这些内容可以由应用提供或由用户生成。

Core Spotlight 提供了一个 API，通过使用一个应用内唯一的 ID 让内容可被检索，同时可对其执行更新或删除操作。一旦建立了索引，数据就可以通过 Spotlight 和 Safari 被搜索。如果这项数据关联了一个 `NSUserActivity`，那么它也能在公开的索引中生效。

该框架通过结构化数据提供更多的控制方式。特别是，对一个唯一应用 ID 的要求可以使在特定应用中强大的内容搜索更加顺畅。[5]

例 13-5 中的代码展示了如何使用 Core Spotlight 框架。

例 13-5　使用 Core Spotlight

```
//添加内容到索引
#import <CoreSpotlight/CoreSpotlight.h> ❶
#import <MobileCoreServices/MobileCoreServices.h> ❷

-(void)addToIndex:(...) { ❸
    CSSearchableItemAttributeSet *attrs = [[CSSearchableItemAttributeSet alloc] ❹
        initWithItemContentType:(NSString *)kUTTypeText]; ❺

    attrs.title = @"Mango";
    attrs.contentDescription = @"King of Fruits";
    attrs.keywords = @["mango", "fruit", "vegetation"]; ❻

    CSSearchableItem *item = [[CSSearchableItem alloc]
        initWithUniqueIdentifier:@"mango"
        domainIdentifier:@"com.mydomain.item-domain"
        attributeSet:attrs]; ❼

    [[CSSearchableIndex defaultSearchableIndex] ❽
        indexSearchableItems:@[item] ❾
        completionHandler:^(NSError *e) { ❿
            if(e) {
                //错误处理
             } else {
                //一切正常
             }
     }];
}

//在AppDelegate中处理搜索结果链接
-(BOOL)application:(UIApplication *)application
    continueUserActivity:(NSUserActivity *)userActivity
    restorationHandler:(void (^)(NSArray *))restorationHandler { ⓫

    if([CSSearchableItemActionType isEqualToString:userActivity.activityType]) { ⓬
        NSDictionary *details = userActivity.userInfo; ⓭
        NSString *itemId = [details
                objectForKey:CSSearchableItemActivityIdentifier]; ⓮

        //使用唯一ID处理
    }
```

注 5：与 `NSUserActivity` 相比，这里没有要求唯一的 ID。开发者也许会将其放入 `userInfo` 字典，不过这个操作是可选的。

```
        return YES;
    }
```

使用 Core Spotlight 接口需要导入 CoreSpotlight.h❶ 头文件。

使用 UTI 类型常量需要导入 MobileCoreServices.h❷ 头文件。

本例中使用辅助方法 `addToIndex`❸ 将内容添加到索引。为简便起见，在此省略该方法的参数。

`CSSearchableItemAttributeSet` 类 ❹ 可用来定义与索引内容相关的属性。比如，`title`、`contentDescription` 这两个重要的属性定义搜索结果的输出。图 13-2 展示了示例中 Spotlight 关于内容的搜索结果。根据条目的类型，你可以设置一个或多个其他可用的属性。[6]

在本例中，该项内容类型设置为纯文本 ❺。[7]

添加 `title` 和可选的 `contentDescription`❻ 非常重要，它们控制了搜索结果的展示效果（见图 13-2）。

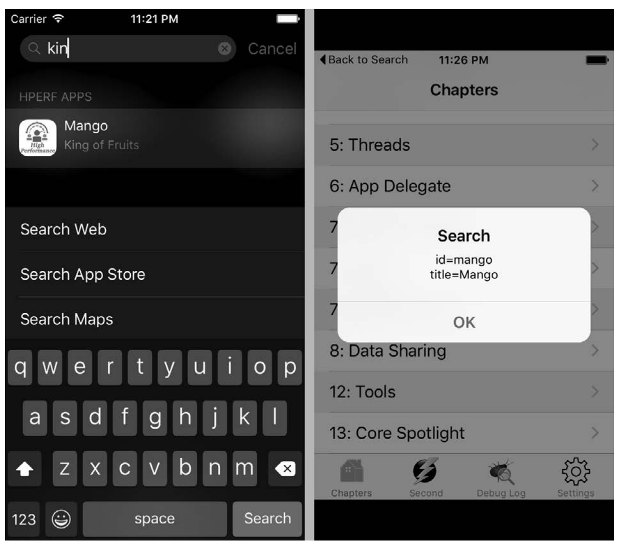


图 13-2：Spotlight 搜索（左）与应用展示的搜索结果（右，演示用的警告弹框）

一旦配置好属性集合，就可以使用它们来让最终的搜索项 `CSSearchableItem` 对象可被索引 ❼。创建时需要一个用于应用的 `uniqueIdentifier`。`domainIdentifier` 的确切目的在编写时还不清楚。[8]

注 6：参见苹果开发者网站的完整列表（http://apple.co/1SvhU5y）。

注 7：我们在 8.3 节中讨论过文档类型。

注 8：它可以仅仅是元数据，也可以对应用中的一项做进一步分类，以便唯一标识符基本上是对单独应用的单独领域生效。这将允许多个 SDK/ 组件在一个应用中无缝工作。

`CSSearchableIndex`❽ 是帮助创建索引的主要的类。`defaultSearchableIndex` 是设备上的全局索引。

`indexSearchableItems:completionHandler:` 方法允许异步检索多个项 ❾。

在完成处理回调 ❿ 中可获得索引结果，完成处理回调不能为 `nil`。

另一方面，当用户点击来自应用的搜索结果时，应用委托方法 `application:continueUserActivity:restorationHandler:`⓫ 将被调用。

`activityType` 在此示例中总是 `CSSearchableItemActionType`。

⓬`userInfo`⓭ 字典至少需要存储一项。该项必须使用 `CSSearchableItemActivityIdentifier`⓮ 的键来提供之前创建 `CSSearchableItem` 使用的唯一 ID。使用该值来查找应用中对应的资料，并展示细节。

当使用 Core Spotlight 接口时，你应当遵循以下与性能相关的建议。

- 在 `title` 和 `description` 中提供充足的信息，但不要使它们过于冗长，否则不但用户无法看全数据，而且会影响序列化 / 反序列化的耗时。
- 不要过度使用 `keyword`，除了可能招致惩罚，还会影响索引创建和搜索性能。
- 尽可能减少用户活动的 `userInfo` 字典中提供的内容。只提供快速展示特定结果所需的数据。

3. Web标记

从 iOS 6 起，如果用户在 iOS 设备 Safari 移动浏览器中访问应用网页，智能应用广告条[9]已经成为提高应用下载量的标志方法。

苹果公司已经引入了新的增强功能。Applebot（https://support.apple.com/en-us/HT204683）将会检索网页寻找元标签，以便为 Spotlight 和 Siri 搜索使用的公共索引提供数据。 在 2015 年的全球开发者大会上，苹果公司提到它将支持标准元标签，[10]其中包括但不限于在 Schema.org 上定义的，以及遵循 Open Graph 协议的标签。在这些公开标准还不能满足需求的特定场景下，苹果公司已经配置了新的标签。

对于在搜索中展现的应用内容，你可以使用这些标签来提供特定的页面展示。

13.1.3　搜索最佳实践

虽然通用链接和搜索看起来很不一样，但二者其实是相辅相成的。`NSUserActivity` 和 Core Spotlight 使用本地内容建立索引，并且也可以向苹果服务器提供。当网页标记能够链接到应用时，它使用 Applebot 帮助苹果服务器建立索引。

但是，这些结果只会在满足一定人气度和其他因素时出现。绝大多数的其他因素目前还不明确，但是开发者必须提供充分的信息来充实索引。这将确保用户可以发现你的应用和其中的内容，并与其交互。

注 9：iOS Developer Library,“Promoting Apps with Smart App Banners”(http://apple.co/1NnawJc).

注 10：iOS Developer Library,“Mark Up Web Content”(http://apple.co/1qMnZ7L).

以下列表包含了我的一些初步建议。（注意，此列表中的最佳实践并未经过实战检验。它是不全面的，随着我们进一步采用新技术和研究埋点数据，它会不断改进。）

- 使用通用链接。网页和应用应当对同一个链接展示相同的内容。
- 对同一内容建立索引时使用同一 ID，这将确保对相同内容获得更好的访问排名。
- 使用 `description` 属性提高用户体验。
- 尽可能提供缩略图。
- 明智地选择关键词。
  目前还没有资料表明使用多少个关键词会导致排名惩罚，但有充分的理由推测对关键词的滥用会招致惩罚。
- 以有限状态机的方式来实现应用，确保引导到应用的搜索结果可以被优雅地处理。
  用户也许会多次通过相同 / 不同的项或通用链接打开应用。

  如图 13-2 所示，右侧的截图表明，处理结果的应用有一个返回搜索的按钮，以便用户可以返回 Spotlight。但是，此时可获取的通知只有 `application WillResignActive` 以及后续的事件。

  严格来说，这意味着，你不清楚用户是否接到了来电、点击了 Home 按钮，还是按下了返回搜索按钮。

  这也意味着，一旦应用进入后台，你并不知道如果处理上一次的搜索结果或在应用中打开的通用链接——是应当保留打开的状态，还是将应用返回之前打开链接前的状态呢？

  系统提供的返回按钮让用户回到源应用。应用获得 `applicationDidBecomeActive:` 的回调，但是区分不了是用户进行应用的切换还是点击了返回按钮。因此，建议在应用内提供一个自定义的返回按钮，这样一来，当用户返回应用时，上一次的结果仍然可以显示出来，同时也提供了一个回到之前状态的选项。

  下一个要处理的复杂问题是，如何管理链接到应用的多个结果 / 链接。在用户进行多次返回操作的过程中，应用也许就已经结束而看不到原来的状态。

  是否在*回退栈*中支持多项调用取决于产品决策。

  当多个项存在于回退栈时，是否提供一个直接链接到达应用原先状态同样取决于产品决策。

  工程师将必须支持这项功能。

13.2 用户界面

iOS 9 有大量关于用户界面层的更新。无论你是否使用它们都将会影响你的应用的性能。为了讨论方便，我们将它们分为两大类：

- UIKit 框架的改动
- Safari 服务框架的改动

游戏相关的框架，如 GameplayKit、Model I/O、MetalKit、Metal、SceneKit 和 SpriteKit 框架都不在讨论范围。

13.2.1　UIKit框架

iOS 9 提供了一个新的容器视图——UIStackView（http://apple.co/1Noq1jW），用于帮助开发者在水平或垂直方向渲染多个视图。

当你需要创建一个表单风格的界面，在水平或垂直方向排列多个视图时（这种情况似乎更常见），UIStackView 将会为你提供一个更快捷、更方便的途径。

回想之前讨论过的邮件视图（见 6.2.6 节中的图 6-11），它有七个子视图。使用自动布局意味着多个约束需要小心的手工处理，以及解线性方程组时的运行时的额外开销（参见 6.3 节中的图 6-14）。

使用自动布局的开销不仅体现在运行时，也体现在设计时。图 6-11 有超过 20 个约束。手工设置它们，让其在 iPhone 和 iPad 的不同屏幕上生效绝非易事。

借鉴安卓

UIStackView 是一项广受欢迎、让人翘首以盼的功能。

安卓在早期就有了 LinearLayout（http://developer.android.com/reference/android/widget/LinearLayout.html），很高兴苹果团队能向其他团队学习，从而为应用开发人员提供方便，最终惠及终端用户。

在雅虎公司，我们实现了自己版本的线性布局，并获益良多。对于拥有 10~12 个 UI 元素的视图，运行时创建和布局时间减少了 19%。对 UITableView 中循环使用的视图，重新布局也减少了超过 10% 的时间。

在开发阶段，一个复杂的视图设计将花费三天来满足跨 iPhone 和 iPad 的使用，但使用新组件只需要花费几个小时。

Xcode 7 的故事板编辑器支持将多个视图推入 UIStackView。如图 13-3 所示，你可以在工具栏底部偏右的位置看到一个新图标。选中多个视图，然后点击这个入栈形状的按钮。所有选中的视图都将被推入一个新的 UIStackView。

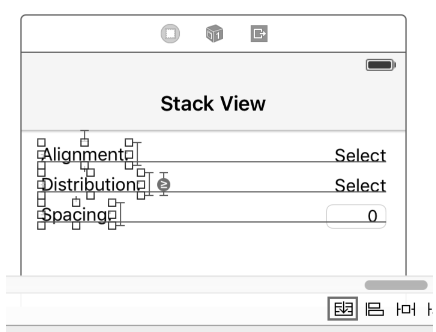


图 13-3：故事板——在 UIStackView 中排列多个视图

以下属性控制了 UIStackView 中受管理视图的渲染效果。

- axis

  定义了布局的方向，可以是 UILayoutConstraintAxisHorizontal 或 UILayoutConstraintAxisVertical。

  UILayoutConstraintAxisVertical 表示受管理的视图将会沿垂直方向进行渲染。后添加的视图将会在前一个渲染好的视图下方。UILayoutConstraintAxisHorizontal 表示受管理的视图将会沿水平方向进行渲染。后添加的视图将会在前一个渲染好的视图右侧。

  默认值是 UILayoutConstraintAxisVertical。

- alignment

  控制视图的对齐方式。与 UIStackView 垂直坐标轴对齐，默认值是 UIStackViewAlignmentFill。

- distribution

  控制视图的大小。该值影响的大小和位置沿着 UIStackView 的坐标轴分布。默认值是 UIStackViewDistributionFill。

- spacing

  以 pt 为单位声明了相邻两个视图的间隔距离，默认值为 0。

- baselineRelativeArrangement

  控制了受管理视图间的垂直间距是否根据基线测量得到。如果是 Yes，那么间距是从一个文字视图的文字最后一行到其后面视图的文字第一行计算。

- layoutMarginsRelativeArrangement

  决定了 UIStackView 平铺受管理的视图时是否参照了它的布局边距或边界。默认值是 No，表示使用边界。

图 13-4 展示了这些属性在横轴上的关联性。对于纵轴而言，只须将这些属性的关联轴旋转 90 度。

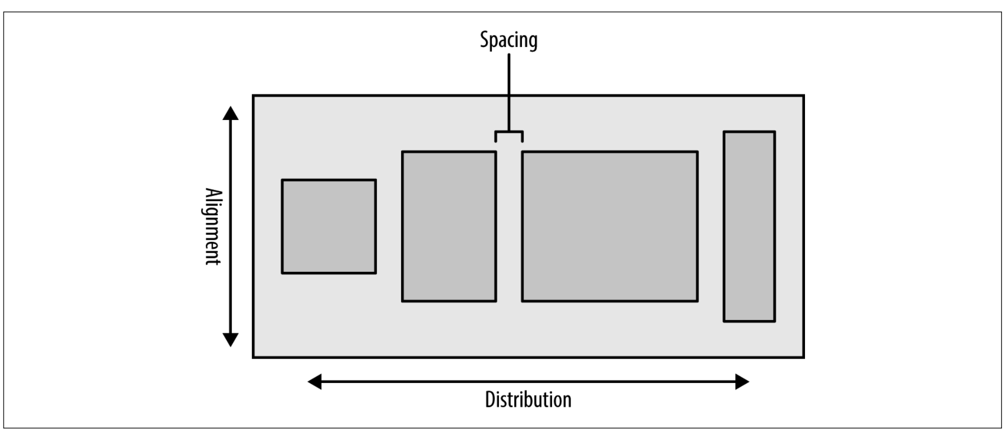


图 13-4：UIStackView——属性及其关联性（横轴）

UIStackView 是 UIView 的非渲染型子类。与 UIView 的其他子类不同，它不在画布上进行渲染。因此，如果重写 drawRect: 方法，它对最终效果并没有任何影响。改变 backgroundColor 等其他属性也是一样的结果。

UIStackView 使用 arrangedSubview+s 的概念来布局，使用 +addArrangedSubview 而不是 addSubview 来添加视图到布局当中。

或者，你可以使用 insertArrangedSubview:atIndex: 在非末尾位置插入视图。

在 UIStackView 中使用 removeArrangedSubview: 删除一个视图，同时在子视图被移除时调用 removeFromSuperview。

图 13-5 展示了 XCode 编辑器中 UIStackView 对应的属性。

到目前为止，UIStackView 相关的最佳实践如下。

- 尽可能使用它，这样不仅能减少设计视图的时间，还能提高运行时的性能。
- 使用上述属性控制最终的布局。
- 如果对它的性能不满意，可能是因为视图过于复杂了，那么就使用自定义布局。

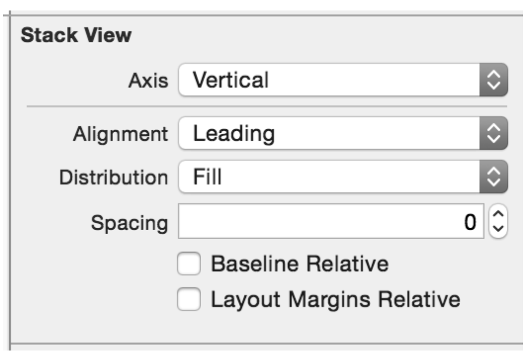


图 13-5 UIStackView——Xcode 属性编辑器

13.2.2 Safari服务框架

iOS 7 中增加了 Safari 服务框架，该框架不为人知是因为其提供的接口主要是为了添加 URL 到用户的 Safari 阅读列表。本节将讨论在 iOS 9 中新增的 SFSafariViewController。在过去，应用大多会使用 UIWebView 来渲染网页内容（我们在 6.2.5 节中讨论过）。iOS 8 引入的 WebKit 框架提供了更高性能的 WKWebView 类。有些应用便从 UIWebView 迁移到 WKWebView。

虽然性能有所提高，但有些问题仍然没有解决。

- 渲染引擎总是基于一个非最新版本的 WebKit（https://www.webkit.org），至少落后于设备上的 Safari 浏览器好几个版本。这意味着在 Safari 中浏览网页内容总是优于使用 UIWebView 和 WKWebView 内置浏览器。
- 没有共享 cookie 的方法。因此，如果用户已经使用浏览器（Safari、Chrome 或者其他）登录了一个网站，在使用内置浏览器时仍然需要重新登录。这就只有两个选项：要么让

用户离开应用到 Safari 或 Chrome，要么让用户重新登录。图 13-6 展示了 HipChat 应用的设置选项。

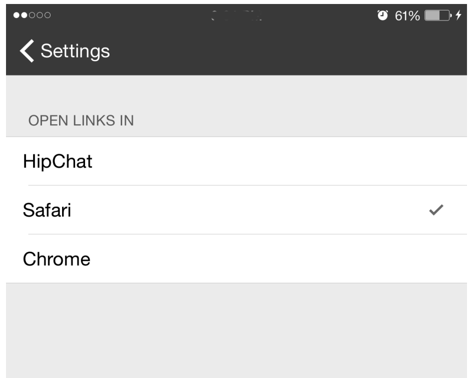


图 13-6：HipChat 打开链接的选项

`SFSafariViewController` 尝试解决这些问题。注意，它不是一个视图而是一个视图控制器。这意味着你不能控制 UI 控件，如地址栏和屏幕下方围绕 HTML 内容的动作按钮，如图 13-7 所示。

视图控制器使用 Safari 浏览器的 cookie，并在单独的进程中运行。这意味着，如果用户使用 Safari 登录过，浏览操作就能无缝进行。

> 应用的 cookie 并不共享给 Safari 视图控制器，这意味着，如果用户仅在应用中登录过，那么还需要再次登录。
> 一旦用户登录，无论是使用视图控制器还是 Safari 应用，会话都将继续下去。

例 13-6 展示了使用 `SFSafariViewController` 的示例代码。

例 13-6　使用 `SFSafariViewController`

```
-(void)showURL:(NSURL *)url {
    SFSafariViewController *safari = [[SFSafariViewController alloc]
        initWithURL:url entersReaderIfAvailable:NO];

    safari.delegate = self;
    [self presentViewController:safari animated:YES completion:nil];
}

-(void)safariViewControllerDidFinish:(SFSafariViewController *)controller {
    //用户点击完成按钮
    [controller dismissViewControllerAnimated:YES completion:nil];
}
```

图 13-7 展示了 Safari 视图控制器的演示效果。

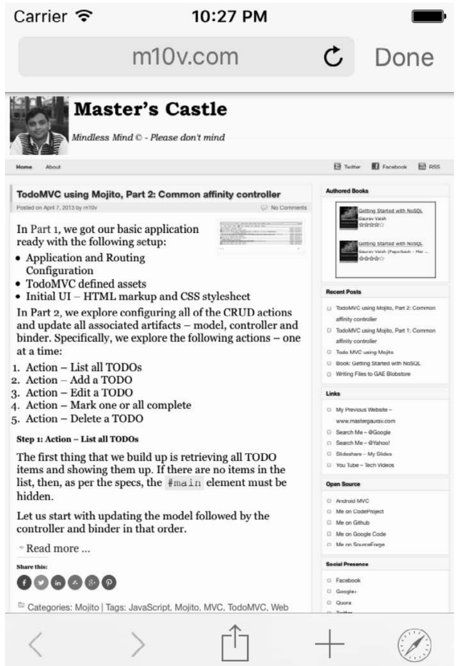


图 13-7：Safari 视图控制器

注意，Safari 浏览器的会话不会被带到 `SFSafariViewController` 视图控制器中（见图 13-8 所示）。

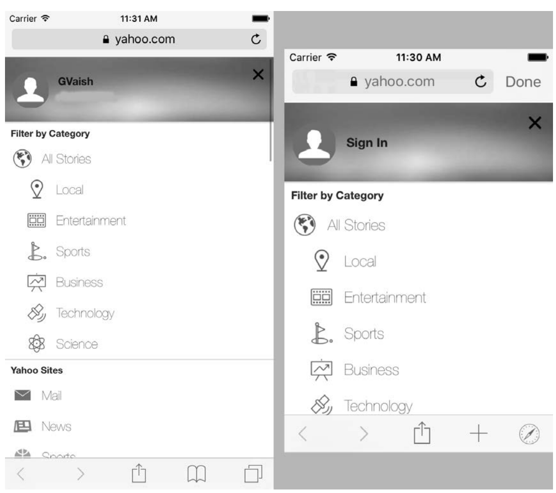


图 13-8：即便用户在 Safari 应用中登录了（左），他在使用 `SFSafariViewController` 的应用中并不会自动登录（右）

我推荐使用 `SFSafariViewController`。但是，因为它只在 iOS 9 中可用，所以你仍然需要对 iOS 7 的用户提供 `UIWebView`、对 iOS 8 的用户提供 `WKWebView`。

使用视图控制器时不要启用 `entersReaderIfAvailable`。让用户来决定是否进入阅读模式。

13.3 扩展

iOS 9 引入了两个新的扩展项（见图 13-9），这两个扩展项会影响用户与应用的交互以及对用户展示的 UI。

- 内容拦截扩展
  在使用 Safari 或 `SFSafariViewController` 时允许限制内容展示。
- Spotlight 索引扩展
  即便应用不运行时也允许更新设备上应用搜索中的索引。

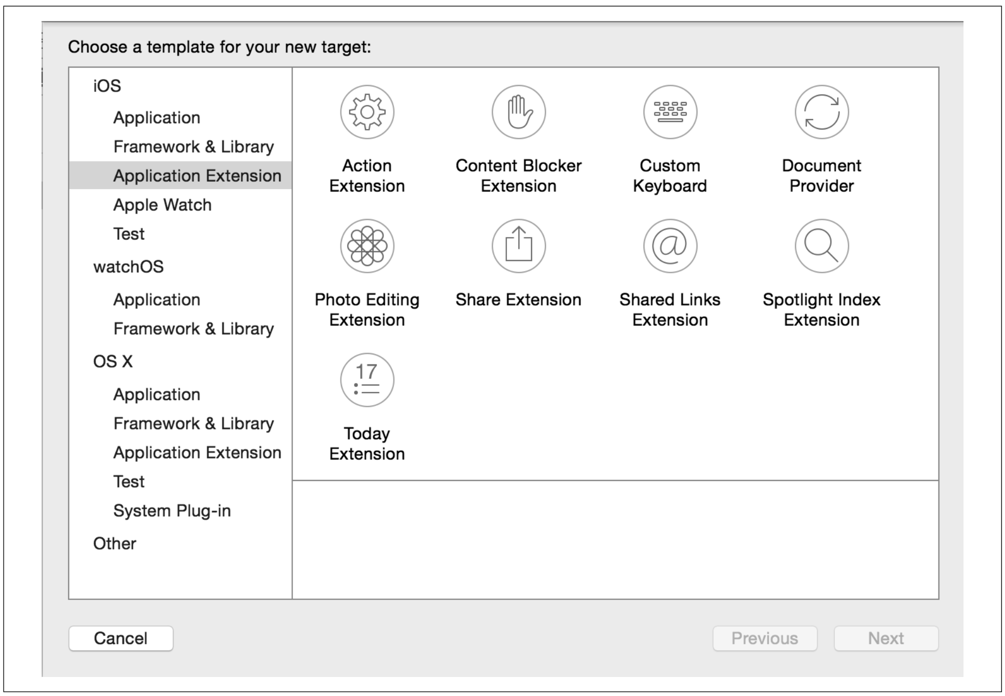


图 13-9：iOS 9 扩展

根据不同的应用，你可能需要实现一个或多个上述扩展。

例如，你可能会创建一个需要保护儿童隐私且限制浏览成人内容的应用。内容拦截扩展就是在设备的 Safari 浏览器中实现如此插件的一个可行方案。

同理，当不再使用时允许更新被索引的内容也是一个好主意。这样可以保证 Spotlight 搜索中不会出现过期的结果。而 Spotlight 索引扩展将帮助你完成这些。

13.3.1　内容拦截扩展

不知你是否在桌面电脑上使用过 Adblock Plus（https://adblockplus.org）来过滤烦人的广告？如果用过，那么你很可能希望移动设备也有此功能。内容拦截扩展可以帮助实现这个功能。

该扩展与 Safari 集成（http://apple.co/1SvqjpG）并允许过滤浏览网页的内容。

如果在系统设置中打开 Safari，你将会发现一个新的内容拦截入口。你可以在设备上可以安装一个或多个内容拦截器并有选择地启用它们。

内容拦截扩展使用一个 JSON 格式的配置文件，从而控制哪些元素可见，哪些需要过滤。JSON 内容的规范可在 WebKit 网站上查看（https://webkit.org/blog/3476/content-blockers-first-look/）。

例 13-7 展示了内容拦截扩展的一个示例。它要求一个类来实现 `NSExtensionRequestHandling` 协议。该协议仅有一个方法 `beginRequestWithExtensionContext:`，它调用 `-[NSExtensionContext completeRequestReturningItems:completionHandler:]` 方法，传入定义好的过滤器。

例 13-7　内容拦截扩展

```
//代码
- (void)beginRequestWithExtensionContext:(NSExtensionContext *)context {

    NSURL *url = [[NSBundle mainBundle]
            URLForResource:@"blockerList" withExtension:@"json"]; ❶
    NSItemProvider *attachment = [[NSItemProvider alloc]
        initWithContentsOfURL:url]; ❷

    NSExtensionItem *item = [[NSExtensionItem alloc] init];
    item.attachments = @[attachment]; ❸

    [context completeRequestReturningItems:@[item] completionHandler:nil]; ❹
}
```

❶ 过滤器定义（JSON 内容）的 URL。

❷ 用 `initWIthContentsOfURL` 初始化器创建一个 `NSItemProvider` 实例。

❸ 创建一个带有 `NSItemProvider` 附件的 `NSExtensionItem` 对象。

❹ 最后，通过调用 `completeRequestReturningItems` 过滤内容。参见例 13-8 了解过滤器定义示例。

例 13-8　过滤器定义示例

```
[{
    "trigger": {
        "url-filter": "webkit.org/images/icon-gold.png" ❶
    },
    "action": {
        "type": "block" ❷
    }
}]
```

❶ 此示例基于将处理的 URL 决定过滤器何时触发。
❷ 当触发器条件满足时应该采取的动作是拦截内容，不展示出来。

图 13-10 展示了如何配置内容拦截器。

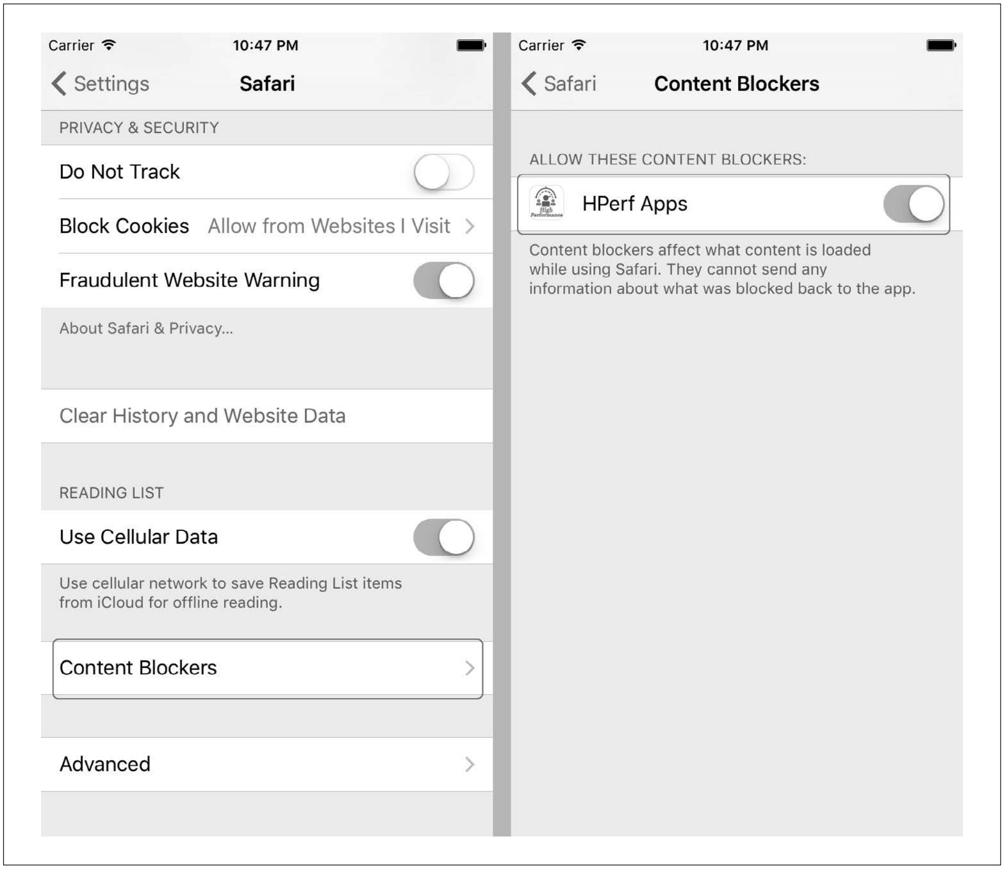


图 13-10：Safari 内容拦截器设置

图 13-11 展示了内容拦截器如何影响一个给定网页的显示效果，图中所用网页是 http://www.webkit.org。左侧截图是内容拦截器打开时的效果，右侧是关闭时的效果。注意，当内容拦截器打开时，WebKit 应用的标识不会展示出来。

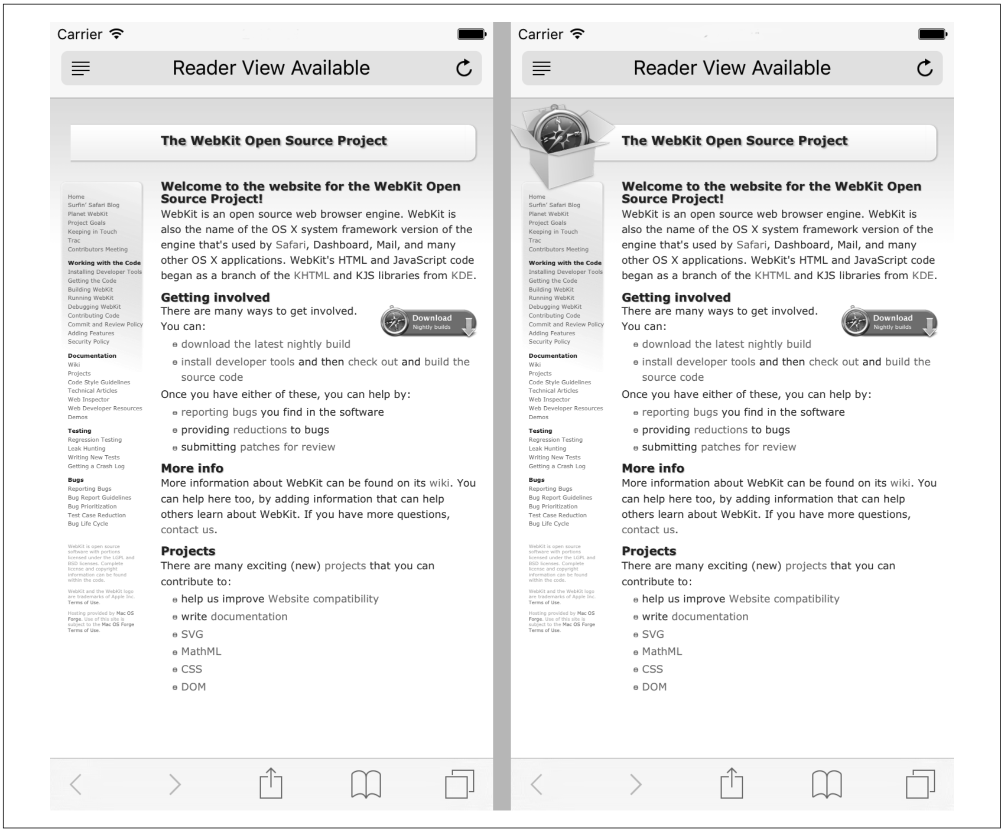


图 13-11：在 WebKit 网站开启内容拦截器的效果（左）与关闭的效果（右）

因为移动端 Safari 第一次支持扩展，所以真实的最佳实践还没有出现。以下是一些基本的建议。

- 每次启用时尽可能使用本地文件系统的文件，而不是从网络上同步。
- 对于动态的过滤器（例如，Adblock Plus 可能会从服务器实现与更新[11]），使用后台下载，以便与服务器周期性地同步。
- 最小化过滤器条目的数量，这在对复杂网页进行过滤时会缩小解析时间。
- 不要滥用过滤。强烈推荐在网站公开该文件，并且让应用可以访问（提示：使用 `openURL:` 方法）。

13.3.2　Spotlight索引扩展

一般来说，你会在应用打开时更新 Spotlight 搜索的索引。通过 Spotlight 索引扩展，你可以创建一个维护索引的扩展，让操作系统调度不在运行状态的应用，给予其一个更新索引及

注 11：Sebastian Noack，“Adblock Plus and (a Little) More”（https://adblockplus.org/blog/content-blocking-in-safari-9-and-ios-9-good-news-or-the-death-knell-of-ad-blocking-on-safari).

验证索引项有效性的机会（例如，验证该项是否可用且未过期）。

使用 Spotlight 索引扩展需要一个继承自 `CSIndexExtensionRequestHandler` 的类并实现以下方法。

- `searchableIndex:reindexAllSearchableItemsWithAcknowledgementHandler:`
  调用该方法以触发索引设备中所有的项。
- `searchableIndex:reindexAllSearchableItemsWithAcknowledgementHandler:`
  调用该方法来触发验证指定唯一标识符的项的有效性。

实现一个 Spotlight 扩展的最佳实践与 13.1.3 节中所讨论的内容相同。

13.4　应用瘦身

在 iOS 9 之前，当需要一个统一的二进制文件来支持多个设备（iPod、iPhone、iPad，以及如今的 Apple Watch）时，你需要将所有的资源打包在一块。这可能会让用户在首次启动下载资源文件时感到不满。

资源文件目录（参见 6.2.3 节）已被证明是一项非常有用的功能，不仅能加快图片加载，还能帮助管理同一个图片针对各种设备的不同版本。然而，这也使得用户最终下载的二进制文件变得十分庞大。

另一方面，优化资源（图像、音频剪辑、视频剪辑）的使用意味着将它们分为两类。一类是总和应用打包在一块，另一类是之后需要时再从服务器下载。这个方法可以减少一些二进制文件的大小，同时避免了过度使用网络。

不过，这意味着要编写很多自定义代码，并且要维护服务器以便在需要时可以下载资源。这对支持多个皮肤和主题的应用来说非常痛苦，最受影响的是游戏类应用。

iOS 9 引入了三个功能来处理资源分发的相关问题：

- 分割
- 按需加载资源
- Bitcode

13.4.1　分割

作为开发者，你仍然可以上传统一的二进制文件到 iTunes Connect，包含对所有设备的可执行程序和资源文件。App Store 负责根据应用支持的设备创建多个下载包。

为下载、安装应用的目标设备创建和分发变种应用包称为分割。

变种包包含特定的可执行体系结构和目标设备需要的资源文件。

- 只下载针对处理器体系结构的可执行文件。
- 按设备支持能力切分 GPU 资源。
- 根据设备类型和分辨率切分图像。

App Store 为安装了 iOS 9 及更新版本的设备发送分割过的包。对之前的版本还是分发统一的二进制包。分割过的图片必须位于资源文件目录下。其他位置的图片不会被切分。

13.4.2 按需加载资源

按需加载资源（http://apple.co/1YrrsmK）指的是 App Store 上与下载的应用包分离的应用内容。

从 iOS 9 开始，你可以对特定资源（如图片和音频剪辑）加标签，然后通过标签来管理这些资源。

具体来说，你可以配置：

- 随应用打包的资源
- 应用第一次启动后安装的资源
- 根据一个关键词安装的所有资源
- 根据一个关键词删除的所有资源

这些资源也可能被分割。这样能保证资源总是应用必需的，而不会占用磁盘空间。设备上包和资源文件变小的副作用是，应用加载时间也会变短。

例如，如果应用有多个皮肤和主题，那么图片包开始可以只包含默认主题。你也可以在设备上保留一些最常用的主题，或者是一个小时前刚浏览过的主题。

图 13-12 展示了从 App Store 到设备按需加载资源的生命周期。

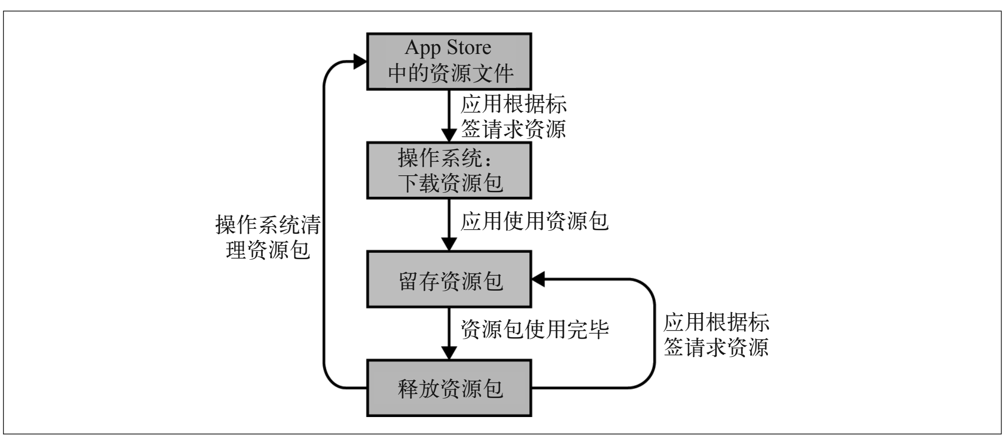


图 13-12：按需加载资源的生命周期

你必须先设置启用按需加载资源的工程，如图 13-13 所示。

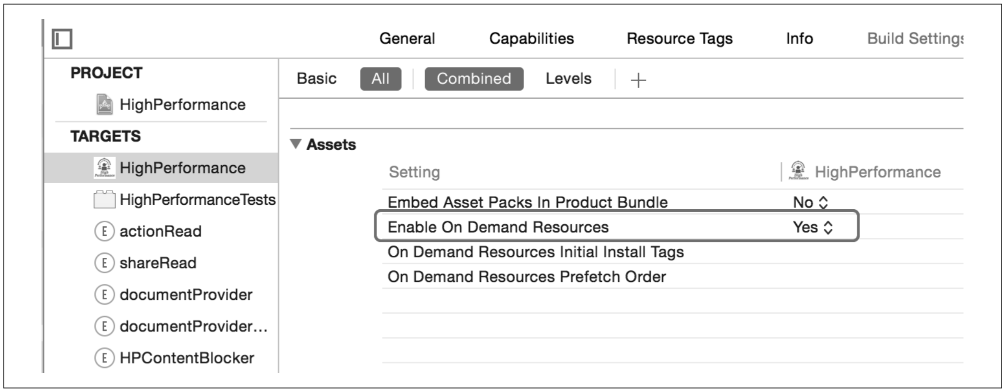


图 13-13：Xcode 启用按需加载资源设置

启用按需加载资源后，下一步是管理标签及其关联的资源。在 Xcode 7 中，工程设置页包含了一个新的 Resource Tags 标签，可用于管理标签（见图 13-14）。你可以使用资源文件目录编辑器来关联标签和资源。

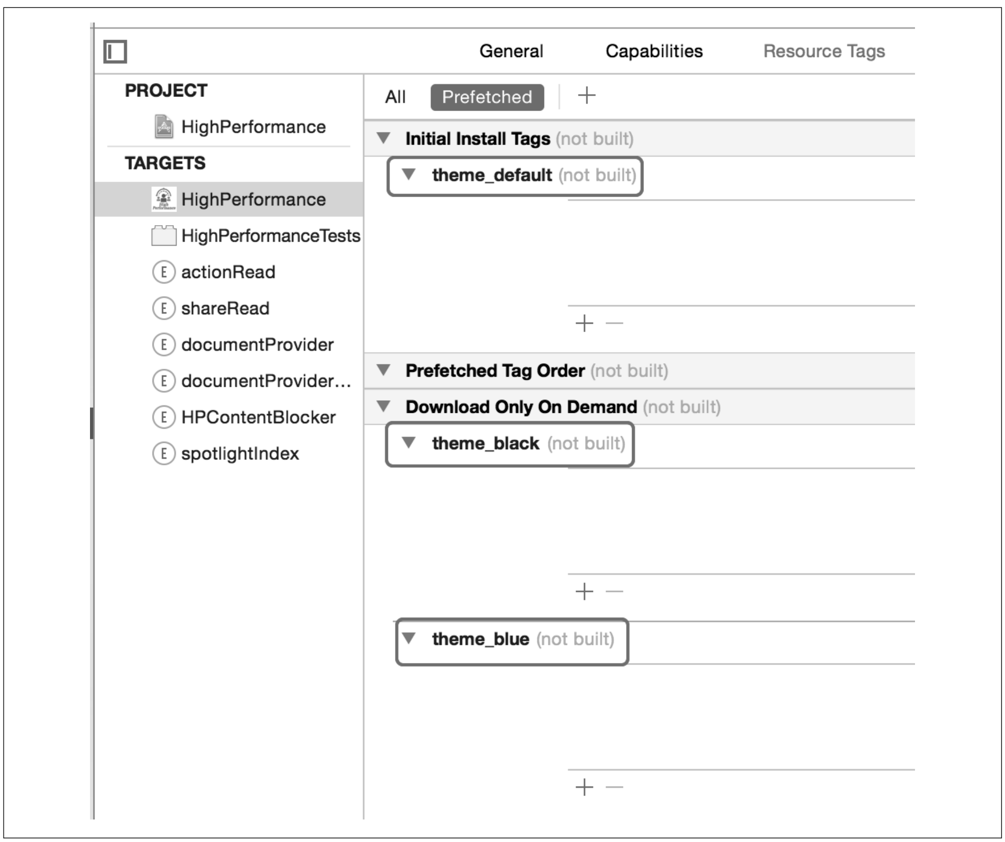


图 13-14：Xcode——Resource Tags 标签

图 13-14 中有三类标签，从名字上区分为 theme_default、theme_black、theme_blue。将 theme_default 标签添加到 Initial Install Tags 集合下，表明该资源将会集成在下载的应用包中。Prefetched Tag Order 里没有标签，这里包含的是在第一次应用启动时下载的资源文件。最后，theme_black 和 theme_blue 放在 Download Only On Demand 集合里，表示这些资源只有在应用需要时才会下载。

图 13-15 展示了如何关联特定的资源对象与标签。

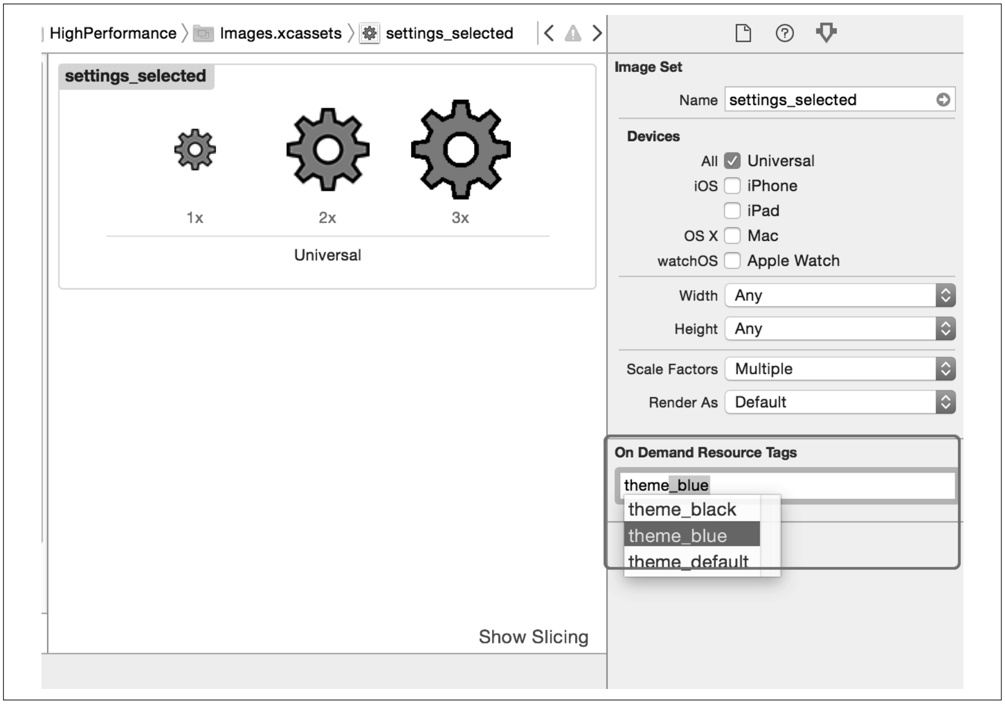


图 13-15：使用资源文件目录编辑器关联标签

一旦完成设置，一切准备就绪。最后一步是管理按需加载资源的标签。

使用类 NSBundleResourceRequest（Khttp://apple.co/1NuyYbs）管理按需加载标签的资源的生命周期。例 13-9 中的示例代码展示了如何下载或停止访问这些资源文件。注意，为了使用这些资源文件，你需要继续使用与之前相同的代码，即 NSBundl 或 UIImage:imageNamed:。

例 13-9　管理按需加载资源标签

```
NSSet *tags = [NSSet setWithArray: @[@"theme_blue"]];
NSBundleResourceRequest *req = [[NSBundleResourceRequest alloc]
        initWithTags:tags]; ❶

[req beginAccessingResourcesWithCompletionHandler:^(NSError *e) { ❷
    if(e) {
        //错误处理
```

```
        } else {
            //正常流程,举例如下
            UIImage *image = [UIImage imageNamed:@"settings"];
        }
    }];

    [req conditionallyBeginAccessingResourcesWithCompletionHandler:^(BOOL available) { ❸
        if(available) { ❹
            //好的,资源已经可以使用,进行处理。
        } else { ❺
            //不可用,也许还未下载或已被清除,现在下载。
        }
    }];

    [req endAccessingResources]; ❻
```

❶ 为应用希望使用的标签创建一个 `NSBundleResourceRequest` 对象。
❷ 下载请求，处理出现错误或下载成功的场景。
❸ 检查资源在设备上是否可用。
❹ 如果可用，那么如前面讨论的那样使用。
❺ 如果不可用，使用 `beginAccessingResourcesWithCompletionHan dler:` 方法排队下载。
❻ 通知系统你已经完成使用给定标签的资源。

13.4.3 bitcode

bitcode 是一个编译好的程序的中间表示形式。

当应用以 bitcode 格式提交到 iTunes Connect 后，将会在 App Store 中以原生格式编译，并且链接到最终的二进制文件。

使用 bitcode 允许苹果公司在未来对应用的二进制文件进行二次优化，而不需要重新提交一个新版本到 App Store。

图 13-16 展示了启用 bitcode 选项的 Xcode 工程设置。新 Xcode 7 工程的默认选项是支持 bitcode。

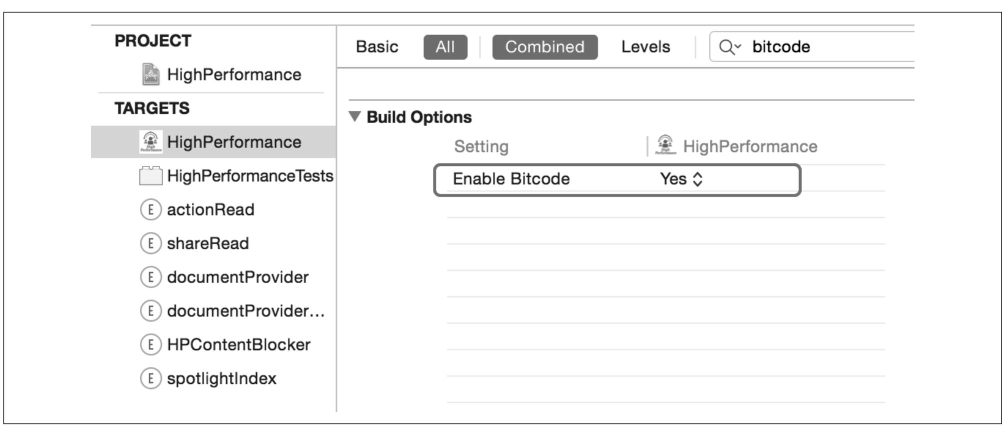


图 13-16：启用 bitcode 选项的 Xcode 工程设置

bitcode 对 iOS 是可选的，但对 watchOS 是必须的。

13.5 小结

iOS 9 拥有一些非常强大的功能，你可以用它们来提高应用的受欢迎程度、底层性能，以及可感知的性能。使用通用链接提供一个统一访问及可分享的 URL，不再需要自定义 URL scheme 来管理令人讨厌的跳转。为应用的公开内容建立索引，以便它们可用于 Spotlight 搜索。理当在应用中使用 `SFSafariViewController`，但需要确保对 iOS 8 或更早的版本提供向下兼容。最后，应用瘦身是一项必须启用的功能，尤其是当应用的资源文件很多，并且在安装后并不需要所有的文件时，你应该按需加载资源。

第 14 章[1]

iOS 10

至此，原书已经告一段落。想必读到此处，你不免有些意犹未尽——止步于 iOS 9 实在是让人心有不甘。一方面，iOS 10 已经面向公众发布了一段时间，截止到 2017 年 1 月 4 日，iOS 10 的比重已经超过 76%[2]；另一方面，iOS 10 是当前最新款的旗舰 iPhone（iPhone 7 和 iPhone 7 plus）唯一可用的操作系统。作为时代最前沿的移动互联网技术开发者，我们怎能对 iOS 10 视而不见呢？

与 iOS 9 相比，iOS 10 带来了更多的新特性，开放了更多的扩展能力，主要体现在以下方面：

- Siri 扩展
- 改进的通知
- VoIP 支持
- iMessage 扩展

这些新特性赋予了第三方应用更强大的能力，也极大地便利了用户的使用。然而，本书的主题是“高性能”。虽然 iOS 10 是当前最新、最优秀的 iOS 版本，但其诸多的新特性却并非与“高性能”主题密切相关。此外，本书中介绍的诸多性能优化技巧通常是基于一些最佳实践和经验总结出来的。因此，本书介绍的性能方面的经验和技巧绝非仅仅适用于某一个或某几个 iOS 版本。这些技巧更像是一棵常青树，即使 iOS 的版本不断地演变与更迭，它们仍然能够永葆青春。

因此，本章介绍的 iOS 10 的相关知识并非本书的重点，更多的只是起到索引的作用，希望能够帮助你更快地入门，并找到相关的学习资料。

注 1：为完善本书内容，跟进最新技术，译者创作了本章。——编者注

注 2：数据来源于 https://developer.apple.com/support/app-store/。

14.1 Siri扩展

伴随着经典的 iPhone 4s，Siri（Speech Interpretation and Recognition Interface，语音交互与识别接口）进入了公众的视野中，时隔 5 年，苹果公司在 iOS 10 版本中对 Siri 进行了有史以来最大的改进：通过开放扩展接口，允许第三方开发者使用。这个扩展接口被称为 `SiriKit`。

`SiriKit` 由两个框架组成：`Intents` 框架用于支持应用和系统之间的基础通信；`Intents UI` 框架提供了展示自定义用户接口的能力。

`SiriKit` 的工作原理如图 14-1 所示：用户“输入”语音，Siri 的音频识别引擎根据词汇表识别出语音表达的意图（Intents），然后通过执行一系列的任务（动作），产生响应。对于应用的开发人员而言，应该关注词汇、应用逻辑和用户界面，其他的事情由 `SiriKit` 为你代劳。

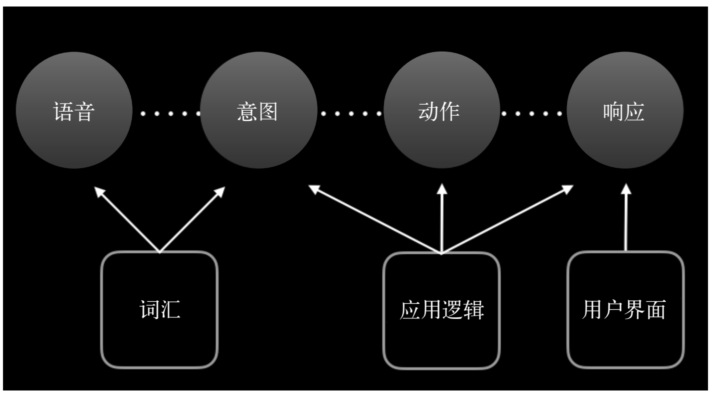


图 14-1：`SiriKit` 的工作原理

就目前而言，`SiriKit` 只支持以下的一些细分领域，因此，只有服务于这些领域的应用能够利用 Siri 的特性进行开发。现阶段支持的领域有：

- 语音和视频通话
- 发送消息
- 收付款
- 照片
- 健身
- 出行预定
- 汽车控制命令（限汽车供应商）
- CarPlay（限汽车供应商）
- 餐厅预定（需要苹果额外支持）

客观来讲，现阶段的 `SiriKit` 能力还比较有限，其扩展性和通用性还不足以被第三方应用广泛使用。尽管如此，`SiriKit` 仍旧是一个良好的开端，其前景不可估量。

使用 `SiriKit` 的步骤如下：

- 首先，创建 Intents 扩展
  - 在 Xcode 中打开应用的工程
  - 然后点击 File > New > Target
  - 选择 Intents Extension，然后点击 Next
  - 输入相关的配置信息，如名称等；如果需要自定义 Siri 的 UI，不妨勾选上 include UI Extension 选项
  - 最后点击 Finish
- 修改 Intents 扩展中的 info.plist 文件
  - 配置 IntentsSuported 键，用于配置需要使用哪些 Intents 服务。

    当前可用 Intents 都隶属于前面介绍过的那些领域。例如，要想支持音频呼叫，请使用 INStartVideoCallIntent。要想了解详细的 Intents 列表，可以参见苹果的官方文档：https://developer.apple.com/library/prerelease/content/documentation/Intents/Conceptual/SiriIntegrationGuide/SiriDomains.html#//apple_ref/doc/uid/TP40016875-CH9-SW2
- 编写处理逻辑

  不同的 Intents 会有对应的处理协议和响应类，例如，INStartVideoCallIntent 对应的处理协议是 INStartVideoCallIntentHandling，而响应类是 INStartVideoCallIntentResponse。通过实现处理协议和创建响应类，开发人员可以实现自己想要的功能。以发送信息为例，需要实现 INSendMessageIntentHandling 协议的方法，并且构造 INSendMessageIntentResponse 类（或子类）的实例，具体见例 14-1。

例 14-1　Intents 处理逻辑

```
- (void)handleSendMessage:(INSendMessageIntent *)intent completion:(void (^)
(INSendMessageIntentResponse *response))completion {

    NSArray<INPerson *> *recipients = intent.recipients; //获取联系人
    NSString* userName = recipients[0].displayName; //这里仅取第一位联系人
    NSString *text = intent.content; //获取消息内容
    NSUserActivity *userActivity = [[NSUserActivity alloc] initWithActivityType:NSS
tringFromClass([INSendMessageIntent class])];
    [userActivity addUserInfoEntriesFromDictionary:@{@"UserName" : userName,
@"Message" : text}];

    INSendMessageIntentResponse *response = [[INSendMessageIntentResponse alloc] in
itWithCode:INSendMessageIntentResponseCodeSuccess userActivity:userActivity];
    completion(response);
}
```

  当 Siri 识别了用户的语音，并识别出用户要使用该应用发送消息时，上述代码就会被触发执行，然后启动该应用。应用会通过 NSUserActivity 获取 Siri 扩展所传递的数据：即用户名和消息内容。
- 向用户申请使用 Siri 的权限

  要想使用 SiriKit，还需要在应用中执行申请授权。你可以在应用运行的任意时刻调用 INPreferences 类的 requestSiriAuthorization: 方法，从而申请使用 Siri 的权限。

有关 SiriKit 的更多信息，可以参见苹果公司提供的官方文档：https://developer.apple.com/library/prerelease/content/documentation/Intents/Conceptual/SiriIntegrationGuide/index.html#//apple_ref/doc/uid/TP40016875-CH11-SW1，以及全球开发者大会中对应的教程：https://developer.apple.com/videos/play/wwdc2016/217/。

14.2　改进的通知

通知系统一直都是 iOS 提供的核心能力，在 iOS 10 版本中，苹果对其作出了重大的改进，主要体现在以下几个方面：

(1) 更加丰富的展示形态
(2) 自定义的交互事件
(3) 改进的 API——新 API 固然好，但大量的旧 API 被标记为 deprecated

14.2.1　申请权限

首先，需要申请使用通知的权限，调用 UNUserNotificationCenter 的 requestAuthorizationWithOptions:completionHandler: 方法即可，具体见例 14-2。

例 14-2　注册申请使用通知的权限

```
[[UNUserNotificationCenter currentNotificationCenter]
  requestAuthorizationWithOptions:
    UNAuthorizationOptionBadge |
    UNAuthorizationOptionAlert |
    UNAuthorizationOptionSound
    completionHandler:^(BOOL granted, NSError * _Nullable error) {
}];
```

14.2.2　触发器

接下来介绍的是 iOS 10 中引入的触发器（Triger）。触发器用于实现在特定条件下触发通知。系统提供了以下 4 种类型的触发器，见图 14-2。

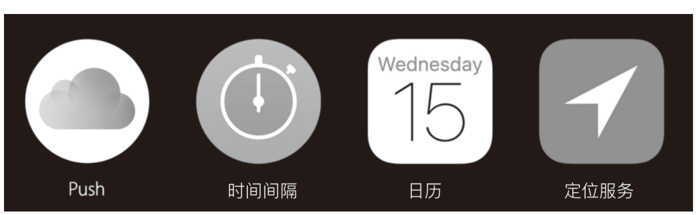


图 14-2：iOS 10 支持的 4 种触发器

- Push：支持远程推送
- 时间间隔：定时触发，支持多次重复触发
- 日历：在某个时间点进行触发

- 定位服务：当用户进入或离开某一区域时触发

触发器使得发送通知变得格外简单，例 14-3 演示了如何使用时间间隔触发通知。

例 14-3　使用触发器推送通知

```
NSString* categoryIdentifier = @"HelloNotificationID"; ❶
//创建要推送的内容
UNMutableNotificationContent *content = [[UNMutableNotificationContent alloc]
init];
content.title = @"iOS 10 Notification";
content.subtitle = @"Trigger by Time Interval";
content.body = @"Hello Notification";
content.categoryIdentifier = categoryIdentifier;
//创建触发器:1秒之后触发,不重复
UNTimeIntervalNotificationTrigger *timeIntervalTrigger = [UNTimeIntervalNotificatio
nTrigger
  triggerWithTimeInterval:1 repeats:NO];
UNNotificationRequest *request = [UNNotificationRequest
  requestWithIdentifier:notificationIdentifier content:content trigger:trigger];
[[UNUserNotificationCenter currentNotificationCenter]
addNotificationRequest:request
  withCompletionHandler:^(NSError * _Nullable error) {
  //完成推送
}];
```

❶ 这里的 `categoryIdentifier` 很重要，后面还会对其进行介绍。

iOS 10 的通知消息中还可以携带附件，目前可以携带图片、音频和视频，但附件有大小限制：

(1) 图片，包括 JPEG、GIF、PNG，最大为 10M
(2) 音频，包括 AIFF、MP3、WAV、MPEG4 音频，最大为 5M
(3) 视频，包括 MPEG、MPEG2、MPEG4、AVI，最大为 30M

具体做法是设置 `UNMutableNotificationContent` 的 `attachments` 属性，代码见例 14-4。

例 14-4　在通知中使用附件

```
NSURL *fileURL = [[NSBundle mainBundle]
  URLForResource:@"picture" withExtension: @"png" ];
UNNotificationAttachment *attachment = [UNNotificationAttachment
  attachmentWithIdentifier:@"aAttachment" URL:fileURL options:nil error:nil];
content.attachments = @[attachment];
```

这里要强调一点，`fileURL` 只支持本地文件 URL，要想展示服务器端的图片或播放远程的音频和视频，我们还需要其他的方法，后面会对此进行详细介绍。

14.2.3　为通知添加交互

通过定义和注册动作（action），你可以为通知的交互展示界面添加交互效果。具体做法是创建 `UNNotificationCategory`，然后向其中添加 `UNNotificationAction` 类的实例，最后将 category 注册到 `UNUserNotificationCenter` 中，具体代码见例 14-5。

例 14-5　注册通知交互

```
UNNotificationAction *action = [UNNotificationAction
  actionWithIdentifier:@"com.app.more" title:@"查看更多"
  options:UNNotificationActionOptionForeground]; ❶
NSString* notificationIdentifier = @"HelloNotificationID"; ❷
UNNotificationCategory *category = [UNNotificationCategory
  categoryWithIdentifier:notificationIdentifier actions:@[action]
  intentIdentifiers:@[]
  options:UNNotificationCategoryOptionCustomDismissAction]; ❸
[[UNUserNotificationCenter currentNotificationCenter]
  setNotificationCategories:[NSSet setWithArray:@[category]]]; ❹
```

❶ 每个 action 代表一个可用于交互的 UI 元素。本例是一个按钮，也可以是一个文本（`UNTextInputNotificationAction`）。

❷ category 的标识符，应与通知内容的标识符一致（见例 14-3）。

❸ 每个 category 可以包含一组 action，以对应某一种通知的多个 UI 元素。

❹ 将 category 注册到 `UNUserNotificationCenter` 中。

已经为通知添加了交互元素（按钮和文本框），显然还需要处理交互的逻辑。在发生交互时，会通过 `UNUserNotificationCenter` 的委托 `UNUserNotificationCenterDelegate` 进行通知。见例 14-6。

例 14-6　处理交互逻辑

```
#pragma mark - UNUserNotificationCenterDelegate
- (void)userNotificationCenter:(UNUserNotificationCenter *)center
  didReceiveNotificationResponse:(UNNotificationResponse *)response
  withCompletionHandler:(void(^)())completionHandler
{
    [[NSNotificationCenter defaultCenter]
      postNotificationName:@"didReceiveNotificationResponse"
      object:nil
      userInfo:@{ @"actionID" : response.actionIdentifier}];
    completionHandler();
}
```

注意，这里的委托方法的参数是 `UNNotificationResponse`，通过它可以拿到很多东西。

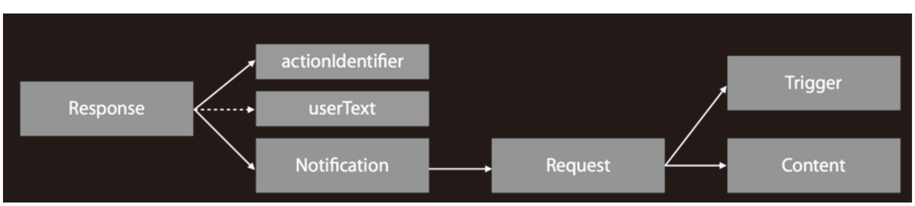


图 14-3：`UNNotificationResponse` 的数据结构

14.2.4　完全自定义展示通知

你是否已经厌倦了千篇一律的通知界面，而想要开发一个个性十足的应用？苹果公司在

iOS 10 版本中提供了这方面的能力。拿售卖火车票的应用来说，如果能在出票成功后向用户推送如图 14-4 所示的效果，那用户体验一定很不错。



图 14-4：自定义通知展示效果

这里需要强调的是，所谓的自定义界面，其实还是有比较大的限制：不可以交互，也不接受触摸事件，仅仅用于信息的展示。

为实现自定义的界面，iOS 10 提供了新的扩展。通过开发通知扩展，第三方的开发人员才可以实现通知的自定义展示方式，具体做法如下：

- 在 Xcode 中打开工程，点击 `File > New > Target`，选择 `Notification Content Extension`
- 输入产品名称等必要的信息，然后点击 `Finish`
- 修改生成的 `info.plist` 文件

`UNNotificationExtensionCategory` 键用于指定通知的 `categoryIdentifier`（不妨回忆一下例 14-3），从而将扩展与某个类别的通知关联起来

当收到通知时，会触发 `UNNotificationContentExtension` 的 `didReceiveNotification:` 方法。具体代码见例 14-7。

例 14-7　响应通知，展示 UI

```
- (void)didReceiveNotification:(UNNotification *)notification {
    //将通知的数据展示在UI之上
    self.label.text = notification.request.content.body;
}
```

如果通知对应的 category 还指定了动作，那么还可以通过 `didReceiveNotificationResponse:completionHandler:` 方法拦截用户交互，参见例 14-8。

例 14-8　响应用户操作

```
- (void)didReceiveNotificationResponse:(UNNotificationResponse *)response
 completionHandler:
 (void (^)(UNNotificationContentExtensionResponseOption))completion
{
    //在此处编写代码,响应用户的操作
    completion(UNNotificationContentExtensionResponseOptionDoNotDismiss);
}
```

14.2.5　通知服务扩展

到目前为止，我们已经介绍了通知系统的主要内容，聪明的你一定会发现一个问题：整个通知系统在能力方面似乎并不完备。通知携带的附件信息只支持访问本地的资源。这意味着其表达能力其实非常的匮乏。毕竟打包在应用中的资源文件既不丰富，也不灵活，很难实现向用户推送丰富多彩的信息。举个例子，要想向用户推送一部新电影的宣传片，却发现通知系统要求相关的视频必须存于用户的手机本地，多么滑稽啊！

幸运的是，苹果提供了有效的解决方案，允许第三方开发者通过*通知服务扩展*来实现类似的需求。*通知服务扩展*为第三方的开发者提供了能力，从而能够将推送的消息进行加工处理（如下载资源），具体做法如下。

- 在 Xcode 中打开工程，点击 `File > New > Target`，选择 `Notification Service Extension`。
- 输入产品名称等必要的信息，然后点击 `Finish`。

与发送远程 Push 的服务端开发人员约定好，将推送的信息标记为“可变”(`"mutable-content" = 1`)，见例 14-9。

例 14-9　服务端推送消息时，标记为可变

```
{
  "aps" : {
    "alert" : {
      "title" : "iOS 10 Push Notification",
      "subtitle" : "Trigger by Push",
      "body" : "Hello Notification"
    },
    "category" : "HelloNotificationID",
    "mutable-content" : 1
  },
  "my-attachment" : "https://example.com/photo.jpg"
}
```

然后，在扩展中对服务端推送的消息内容进行修改。下载其中对应的远程资源，使之成为本地资源，参见例 14-10。

例 14-10　修改推送消息并下载远程资源

```
@interface NotificationService ()
```

```
@property (nonatomic, strong) void (^contentHandler)(UNNotificationContent
*contentToDeliver);
@property (nonatomic, strong) UNMutableNotificationContent *bestAttemptContent;
@property (nonatomic, strong) NSURLSessionDownloadTask *downloadTask;

@end

@implementation NotificationService

- (void)didReceiveNotificationRequest:(UNNotificationRequest *)request
  withContentHandler:(void (^)(UNNotificationContent * _Nonnull))contentHandler {
    self.contentHandler = contentHandler;
    self.bestAttemptContent = [request.content mutableCopy];
    NSString *URLString = request.content.userInfo[@"my-attachment"];
    NSURL* resourceURL = [NSURL URLWithString: URLString];

    self.downloadTask = [[NSURLSession sharedSession]
      downloadTaskWithURL:resourceURL
      completionHandler:^(NSURL * _Nullable location,
        NSURLResponse * _Nullable response,
        NSError * _Nullable error) {
        NSURL *savedURL = 某个可用的本地文件URL;
        [[NSFileManager defaultManager] moveItemAtURL:location toURL:savedURL
error:nil];
        NSError *attachmentError = nil;
        NSString *identifier = [NSString stringWithFormat:
          @"%@%@", URLString, [NSDate date]];
        UNNotificationAttachment *attachment = [UNNotificationAttachment
          attachmentWithIdentifier: identifier
          URL:savedURL
          options:nil
          error: \&attachmentError];
        if (attachmentError) {
            self.bestAttemptContent.body = [NSString stringWithFormat:@"%@
attachmentError:%@", self.bestAttemptContent.body, attachmentError.
localizedDescription];
        } else {
            // 将下载完成的文件添加到content中
            self.bestAttemptContent.attachments = @[attachment];
        }
        self.contentHandler(self.bestAttemptContent);
    }];

    [self.downloadTask resume];
}

- (void)serviceExtensionTimeWillExpire {
    //无法在规定的时间内完成下载,取消下载
    [self.downloadTask cancel];
    self.contentHandler(self.bestAttemptContent);
}

@end
```

苹果对通知服务扩展有一些限制：

- 扩展执行的时间不得超过 30 秒，因此，远程资源必须在 30 秒钟之内下载完毕，否则会先触发 `serviceExtensionTimeWillExpire` 方法，然后中止扩展的执行。

  要谨慎使用远程推送的附件，一定要避免因为数据量过大而导致用户的数据流量费用剧增。你可以使用第 3 章中介绍的知识点，根据用户的实际网络状况（WiFi、4G 或其他），灵活地调整资源下载的逻辑。例如，在使用 WiFi 网络时下载品质更高的音频、视频或图片资源，而在蜂窝网络时不下载或下载品质较低的资源。

14.3 iMessage扩展

iMessage 是苹果公司在 2011 年随 iOS 5 推出的一项免费服务，用户可以通过它利用互联网服务发送文字信息和图片信息。时隔 5 年，伴随着 iOS 10 的出现，这一服务终于得到了重大更新。

通过提供 iMessage 的扩展，苹果允许第三方对 iMessage 进行扩展，这使得 iMessage 有了无限的可能性。官方提供了以下的扩展类型：

- 贴纸（表情包）
- 交互式信息
- 其他内容——照片、视频、文本、链接，等等

同时，iOS 10 还提供了类似 App Store 的商店功能，即“Messages App Store”，用户可以通过这个商店对扩展进行管理。

iOS 提供了一个全新的 `Messages Framework` 框架，通过它可以开发信息扩展应用，标准的步骤如下。

- 创建信息扩展
  苹果就此提供了两种选择：一种方案是仅仅创建独立的扩展，用于在“Messages App Store”中发售；另一种方案是为现有的应用添加一个扩展包，则该扩展会随应用进行发布。

  对于消息扩展本身而言，两者没有本质区别。在 Xcode 中，点击 `File > New > Project`，选择 `iMessage Application`；或者点击 `File > New > Target`，选择 `iMessage Extension`。

- 编写消息扩展的根 ViewControler
  消息扩展的根 ViewController 继承于 `MSMessagesAppViewController`，`MSMessagesAppViewController` 本身也继承于 `UIViewController`，所以使用起来非常轻松。

如果只是想要编写一个发送贴纸的扩展，则可以直接使用 `MSStickerBrowserView` 类。它能够以网格的形式展示贴纸，还支持翻页，参见例 14-11。

例 14-11　使用 `MSStickerBrowserView`

```
@interface MessagesViewController ()<MSStickerBrowserViewDataSource>

@property (nonatomic) MSStickerBrowserView* browserView;
```

```
@property (nonatomic) NSArray<MSSticker*>* stickers;

@end

@implementation MessagesViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    self.stickers = 载入本地贴纸资源,构造MSSticker对象数组
    self.browserView = [[MSStickerBrowserView alloc] init];
    self.browserView.dataSource = self;

    [self.view addSubview: self.browserView];
    self.browserView.backgroundColor = [UIColor lightGrayColor];
}

|#pragma mark - MSStickerBrowserViewDataSource

- (NSInteger)numberOfStickersInStickerBrowserView:
  (MSStickerBrowserView *)stickerBrowserView {
    return [self.stickers count];
}

- (MSSticker *)stickerBrowserView:(MSStickerBrowserView *)stickerBrowserView
  stickerAtIndex:(NSInteger)index {
    return self.stickers[index];
}
```

MSStickerBrowserView 的用法其实大有似曾相识之感，MSStickerBrowserViewDataSource 用于向 MSStickerBrowserView 提供数据源，其用法与 UITableView 格外相似。

此外，MSMessagesAppViewController 还在 UIViewController 的基础上增加了一些特有的生命周期事件和一些常用的属性。其中，最重要的要数 @property MSConversation *activeConversation;。

MSConversation 用于表示一次对话，调用 MSConversation 可以发送消息，具体 API 如下。

```
//会话中插入MSMessage
- (void)insertMessage:(MSMessage *)message completionHandler:(nullable void (^)
(NSError * _Nullable))completionHandler;
//会话中插入发送贴纸
- (void)insertSticker:(MSSticker *)sticker completionHandler:(nullable void (^)
(NSError * _Nullable))completionHandler;
//会话中插入发送文本消息
- (void)insertText:(NSString *)text completionHandler:(nullable void (^)(NSError *
_Nullable))completionHandler;
//会话中插入附件
- (void)insertAttachment:(NSURL *)URL withAlternateFilename:(nullable NSString
*)filename completionHandler:(nullable void (^)(NSError * _Nullable))
completionHandler;
```

苹果公司并没有将信息扩展简单地定义为发送表情包。通过定义 MSMessage，你能够实现更加复杂的交互式消息，参见例 14-12。

例 14-12　使用 MSMessage

```
MSMessageTemplateLayout* layout = [[MSMessageTemplateLayout alloc] init];
layout.imageTitle = @"图片标题";
layout.caption = @"标题";
layout.image = [UIImage imageNamed:@"someImage"];

MSMessage* msg = [[MSMessage alloc] init];
msg.layout = layout; ❶

[self.activeConversation insertMessage:msg completionHandler:^(NSError * _Nullable
error) {
    NSLog(@"%@", error);
}];
```

❶ MSMesaage 的 layout 属性决定了消息的展示布局，目前只支持一种布局，即 MSMessageTemplateLayout，其展示效果如图 14-5 所示。

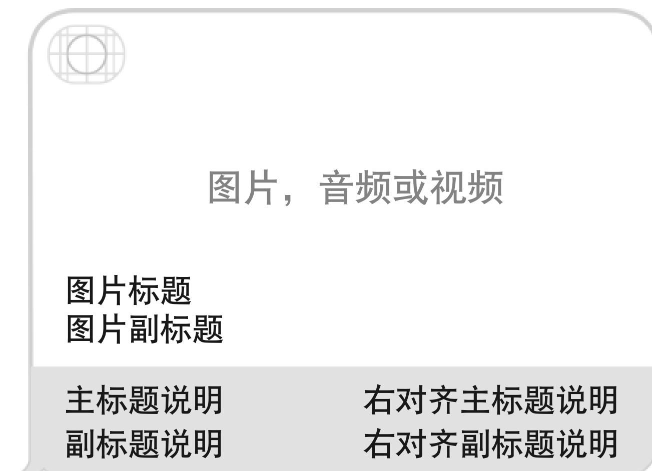


图 14-5：MSMessageTemplateLayout 的展示布局

关于 Messages 框架的详细信息，可以参见苹果的官方文档[3]，以及 2016 年的全球开发者大会中的相关视频[4]。此外，苹果公司还提供了一个完整的示例，演示了利用 Messages 框架与好友合作设计一款冰淇淋[5]。

注 3：Messages: https://developer.apple.com/reference/messages.

注 4：iMessage Apps and Stickers, Part 1: https://developer.apple.com/videos/play/wwdc2016/204/; Part 2: https://developer.apple.com/videos/play/wwdc2016/224/.

注 5：Ice Cream Builder: A simple Messages app extension: https://developer.apple.com/library/prerelease/content/samplecode/IceCreamBuilder/Introduction/Intro.html.

14.4 VoIP支持

iOS 10 提供了一套新的框架——CallKit。利用 CallKit，第三方的即时通信软件可以得到与系统电话应用一致的体验。这对于运营商来说无疑是个沉重的打击。CallKit 提供的能力如下。

- 通知能力：在通话发起时，接收方可以收到通知。这个通知是全屏展示且伴有铃声，体验与系统内置的电话服务一致。
- 使第三方语音电话拥有更高优先级，通话时不被系统来电打断。
- 通话记录：通话过程也会记录在系统电话应用的最近通话中。
- 通话拦截：在来电时，对来电进行阻止或识别。

这里需要强调一点，CallKit 的定位并不是提供通信服务，而是提供系统的入口。真正的通信协议、通信过程需要第三方应用的开发人员自己来实现。第三方应用的开发人员可以通过调用 CallKit 的 API，将通信过程反映到系统的电话应用中。理解这一点后，再学习使用 CallKit 就会容易很多。CallKit 有两个非常重要的类，分别是 CXProvider 和 CXCallController。

CXProvider 用于通知，进而更新系统的状态。CallKit 的核心就是将第三方系统的通信状态报告给系统。CXProvider 有一系列以 reportXXX 开头的方法，用于通知系统当前通信的状态：

- reportNewIncomingCallWithUUID:update:completion: 报告电话呼入状态
- reportOutgoingCallWithUUID:startedConnectingAtDate: 报告电话（连接中）状态
- reportOutgoingCallWithUUID:connectedAtDate: 报告电话（已连接）状态
- reportCallWithUUID:updated: 报告电话更新状态（是否支持视频、是否进行群组通话、是否挂起并保持通话等）
- reportCallWithUUID:endedAtDate:reason: 报告电话挂断状态

举例，当应用收到语音 / 视频呼入请求时，调用 reportNewIncomingCallWithUUID:update:completion: 方法，系统会展示出有电话呼入的界面。

对于唤起的通话界面，用户可以进行一些操作，比如点击“挂断”按钮。这些操作可以通过 CXProvider 的委托 CXProviderDelegate 进行拦截处理。例如，当用户选择接听电话时，会触发 provider:performAnswerCallAction: 方法；而在用户挂断电话时，会触发 provider:performEndCallAction: 方法。

CXCallController 用于发起相关的动作，其中包括：呼叫、结束呼叫、保留等。CXCallController 会通过 CXTransaction 发起不同的动作，每种动作分别对应不同的 action，例如：

- CXAnswerCallAction：接听电话
- CXStartCallAction：开始外呼电话
- CXEndCallAction：结束通话
- CXSetHeldCallAction：保持通话

等等

看到这里，你可能会觉得一头雾水。如果通过 `CXCallController` 发起了通话，接收方是如何收到通知的呢？毕竟 `CallKit` 并不参与通信相关的事情。

事实上，你需要使用 `PushKit`，通过 Push 通道发送和接收通话（来电）通知。而具体到通信过程，需要开发人员开发网络服务，并且调用 `AVFoundation` 对音频或视频进行采集和回放，最终完成整个通信的闭环。

> 一定要谨慎地使用数据流量，要避免因为数据量过大而导致用户的数据流量费用剧增。你可以使用第 3 章中介绍的知识点，根据用户的实际网络状况（WiFi、4G 或其他），及时调整通信音频、视频质量，在用户流量开销和服务质量之间进行权衡。

关于 `CallKit` 的更多知识，可以参见苹果提供的示例[6]。

注 6：Speakerbox: Using CallKit to create a VoIP app https://developer.apple.com/library/prerelease/content/samplecode/Speakerbox/Introduction/Intro.html

作者介绍

Gaurav Vaish 在 12 岁时首次接触到 GW-BASIC，然后就因为它的简洁而爱上了它。在随后的 20 多年中，他使用过大多数的主流语言，在所有流行的操作系统上，甚至在每一种受欢迎的设备上都编写过代码。

他现在就职于雅虎总部的移动和新兴产品团队——具体来说是移动 SDK 小组。这个小组的使命是创建优化的可重用方案，将其整合到雅虎的移动应用中，它们能够在几十种设备上运行，每月有数亿用户使用，每周执行超过十亿次的用户交互，并且每天处理超过十亿次的网络连接。

Gaurav 于 2002 年在 Adobe Systems India 开启了他的职业生涯，就职于工程解决方案部门。2005 年，他成立了自己的公司——Edujini 实验室，专注于企业培训和协作学习。

Gaurav 获有印度 IIT Kanpur 电子工程语音信号处理专业的科技学士学位。

他著有 *Reflections by IITians* 和 *Getting Started with NoSQL*，还管理着自己的私人博客 http://www.m10v.com。

封面介绍

本书封面中的动物是一只中贼鸥，它们是一种广泛出现在世界各地的迁徙海鸟。它们在热带海洋过冬，然后在夏天返回北部，在北极苔原产蛋。虽然它们的名字与波美拉尼亚的波罗的海地区无关，但波美拉尼亚贼鸥是这种鸟常常被叫错的称谓。

成年的中贼鸥身长在 45 厘米至 66 厘米之间，体重近 1 公斤。因为它们与短尾贼鸥（另一种海鸟）十分相似，所以想要辨别该种贼鸥会较为困难。实际上，成年的中贼鸥有多种形态，或者具有三种不同的颜色图案。这三种图案包含棕色、黑色和白色的阴影，但通常有白色的下腹部和白色翅膀。

中贼鸥以鱼、腐肉、小鸟，甚至啮齿动物为食。它们会在飞行途中从海鸥、燕鸥或塘鹅那偷鱼，只会被成年的白尾鹰和金鹰猎食。一旦雌性中贼鸥在北极筑巢，它们会在地面的草巢中孵育两三颗黄褐色的鸟蛋。中贼鸥以护巢性强而著称；虽然它们不能造成较大的伤害，但若有一个生气的鸟妈妈俯冲至你的头部，则确实是个噩梦啊！

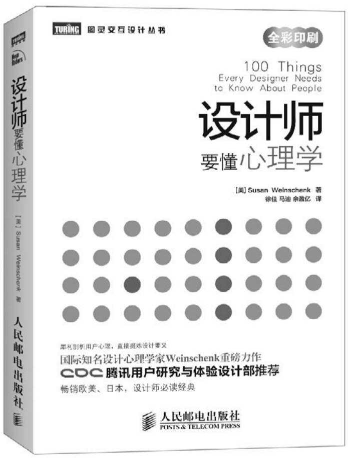
O'Reilly 封面上的许多动物都已濒临灭绝，但它们的存在对世界至关重要。想要了解如何帮助它们，可以登录 animals.oreilly.com。

本书封面图片来自 Lydekker 的 *Royal Natural History* 一书。

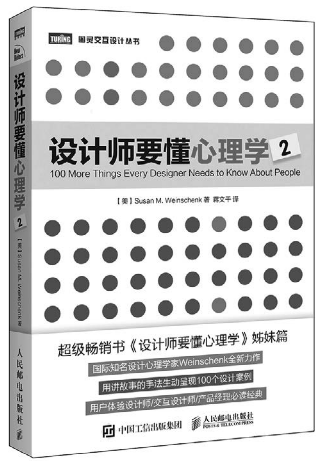
延 展 阅 读

全彩印刷

- 出自国际知名的设计心理学博士Susan Weinschenk之手
- 以创造美观实用的设计为宗旨，讨论设计师必须了解的心理学问题
- 每个问题都配以权威经典示例，并给出即学即用的设计建议



书号：978-7-115-31308-9
定价：49.00 元



书号：978-7-115-42784-7
定价：59.00 元

全彩印刷



- 全球热销超过1 000 000册
- 适用于各行业与文字打交道的人
- 有大师指导，人人都能成为设计师

书号：978-7-115-40440-4
定价：59.00 元（简装）
书号：978-7-115-39598-6
定价：79.00 元（精装）

O'REILLY®

高性能iOS应用开发

本书为有经验的iOS开发者提供构建优异应用移动性能所需的开发建议和最佳实践，帮助读者解决常见性能问题。

作者Gaurav Vaish从工程角度演示了编写最佳代码的方法，详尽介绍如何设计和优化iOS应用，以便在网络较差、内存较低的情况下提供流畅的用户体验。书中还提供了可以反复使用的Objective-C代码，以及一些能够从众多应用中脱颖而出的高性能原生iOS应用。

- **概述跟踪应用性能时需要衡量的参数以及如何衡量性能。**
- **通过最小化内存和功耗以及并发编程来编写高效应用，并探索一些相关选项。**
- **优化应用的生命周期和UI，以及网络、数据共享和安全功能。**
- **了解应用的测试、调试和分析工具，并监控应用。**
- **从真实用户处收集数据来分析应用的使用情况，找出瓶颈并修复。**

Gaurav Vaish 就职于雅虎公司的移动和新兴产品团队，为每月有数亿人使用的移动应用创建优雅的可重用方案。他曾是IIT全球指导计划的成员，还在印度班加罗尔创立了InColeg Learning及Edujini Labs有限公司。

MOBILE

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机 / 移动开发 / iOS

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-45120-0

ISBN 978-7-115-45120-0
定价：89.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks